CoderChrome:
Augmenting source code with software metrics

Matthew Harward
mjh191@student.canterbury.ac.nz
Supervisor: Dr. Warwick Irwin
Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand

November 5, 2009

Contents

List of Figures

Abstract

Software is typically big and complex. Software metrics provide measurements of software products and development processes, in order to help software developers understand and improve their products. Metrics, however, can add to developers' information overload problems, so visualisation techniques are needed to allow large volumes of measurement data to be efficiently communicated to an observer.

Software measurement data is normally presented in reports, tables, or graphical visualisations that are distinct from the primary way developers view their products: in a source code editor. This separation makes it hard for developers to relate measurement data to the features being measured. Additionally, the intrusive task of having to run measurement tools and accommodate different views provides a disincentive for measuring at all. We present a new visualisation technique that directly applies a visualisation overlay to source code. We have developed a tool, CoderChrome, providing this functionality for the Eclipse Java editor.

We discuss our progress in evaluating this visualisation to determine if this approach has the potential to improve the effectiveness of developers. The tool provides a basis for continued research into the usefulness of software metrics and understanding of the best practices of developers.

Acknowledgments

I would like to deeply thank my supervisor Warwick Irwin for his continued and long lasting support. I would also like to thank the members of the Software Engineering Visualisation Group (SEVG) for their invaluable input into this project, in particular Neville Churcher for his much appreciated feedback and assistance. Finally, I would also like to thank Andy Cockburn for his excellent advice on the HCI issues pertaining to this project.

1. Introduction

1.1 Overview

Software engineering is a time consuming and expensive undertaking. While this discipline has advanced dramatically since its inception over four decades ago, many challenges still remain for the software development industry. These challenges consist of issues with the process of development and the product that is being created. From a process perspective, these issues can include those related to the economic realities of software development [10] and the requirements of clients to produce multiple versions [38] of high-quality software to a deadline. It can also include issues relating to the dynamics of large development teams that are often globally distributed [23].

The underlying cause of these process issues, in fact the very reason why software development is a challenge, is the size and complexity of the systems being created. These issues lead to the majority of large software projects being compromised and a significant percentage failing completely [12, 32, 42].

One of the earliest texts that recognise these issues is Brook's "The Mythical Man-Month" [12]. Since this development, a gradual improvement has been provided to software engineering through the development of agile methods [11,24], the shift toward OO languages, the introduction of design patterns [19], and the push towards Test Driven Development (TDD) [6, 9]. While it is clear that the success rates for software development are increasing, more progress is needed [42].

Software engineering draws on conventions established by tradition physical engineering disciplines. One of the key characteristics of these disciplines is the use of quantative approaches to managing both products and development processes [48]. The field of software metrics is now supported by a large body of literature [16, 17, 21, 29, 30]. The most well used metric is Lines Of Code (LOC); however, a huge body of these metrics exist, one well known example is Chidamber and Kemerer's suite of OO software metrics [13].

However, software metrics alone are not an adequate solution. One of the most fundamental problems faced by software developers is information overload, a problem that originates from the size and complexity of software; metrics data can easily compound the information overload problem. Furthermore, this problem is compounded as this metrics information is not presented in the source code view that developers commonly use. In this case, developers are forced to mentally translate tables of measurements into a form that is useful. One possible solution is to use a form of visualisation to display this data.

The majority of a developers time is spent in a source code editor, such as those found in modern IDEs. One major limitation of existing software visualisation techniques is the need for a separate representation outside of this source code editor. This research is an exploration of the idea of *in situ* visualisation, that is augmenting the traditional source code view with metrics information.

Such a technique would allow developers to directly view and comprehend data sets that previously made little sense. In essence, developers need an understanding of the *context* they are working in to use their *focus*, the source code editor, to its full potential. We could show developers the number of edits of any line of code providing them with an excellent tool for identifying potential problems. We could show metrics like the Cyclomatic Complexity of a section of code allowing

developers to easily detect which methods need refactoring. We could use the tool to identify broken heuristics [40] or code smells so that the developer can receive sufficient information to critique their own design. This technique has the potential to be highly beneficial to the productivity and satisfaction of developers. In the next section, our exploration and solution to this problem will be discussed.

1.2 Project Goals

In this project, our main goals are:

- To identify a set of visualisation techniques that can be used to provide metrics information to developers in a source code editor without the need to change a developers focus.
- To produce and document an extensible tool that supports multiple methods of visualisation and a wide range of metrics.
- To ensure that the tool is sufficiently flexible and robust that it may be used in a large commercial software development environment, because the problems addressed by this report are most prevalent in industrial scale software.
- To demonstrate the viability of *in situ* visualisation and clarify the issues surrounding its applicability.

This project is part of a larger research program that aims to improve the usage of metrics in software engineering. The tool produced by this project will provide a valuable platform for the continued evaluation of software metrics.

1.3 Roadmap

In section 2, we explore the background of this area of research. This will include a literature review of related topics as well as an exploration of the existing tools and techniques. The specifications and formalisation of the visualisation will then be discussed in detail in section 3. Following this, the development of a tool which facilitates this visualisation technique will be described in section 4. In section 5, we will present our results and our current progress in evaluating the potential usefulness of this system. In addition, the relevancy of our work is discussed in section 6 and we also present future areas of work in section 7.

2. Background

In this section, we will firstly explore the concept of an Integrated Development Environment (IDE) which is a central consideration of this project. We will then look into the background behind the visualisation of software metrics. Finally, we will then look at existing research and tools that provide information on the augmentation of source code.

2.1 Integrated Development Environments

The majority of source code editors reside inside a IDE. As the *in situ* visualisation we intend to produce augments a source code editor, background information on IDEs is important.

In general, an IDE provides a suite of tools in a single location that help in the development of software. A wide variety of these environments exist. Some of the most commonly used are Microsoft's Visual Studio, Eclipse and CodeGear's JBuilder. Often these tools provide support for only one programming language; however, some IDEs such as Visual Studio and Eclipse provide support for developing in multiple languages.

Most IDEs provide a common set of features. At the highest level, this most commonly includes a source code editor, an inbuilt compiler, inbuilt build tools, and a debugger. Usually these systems also provide a navigator for maintaining the files within a project, inbuilt tools for unit testing and refactoring, and integration with version control systems.

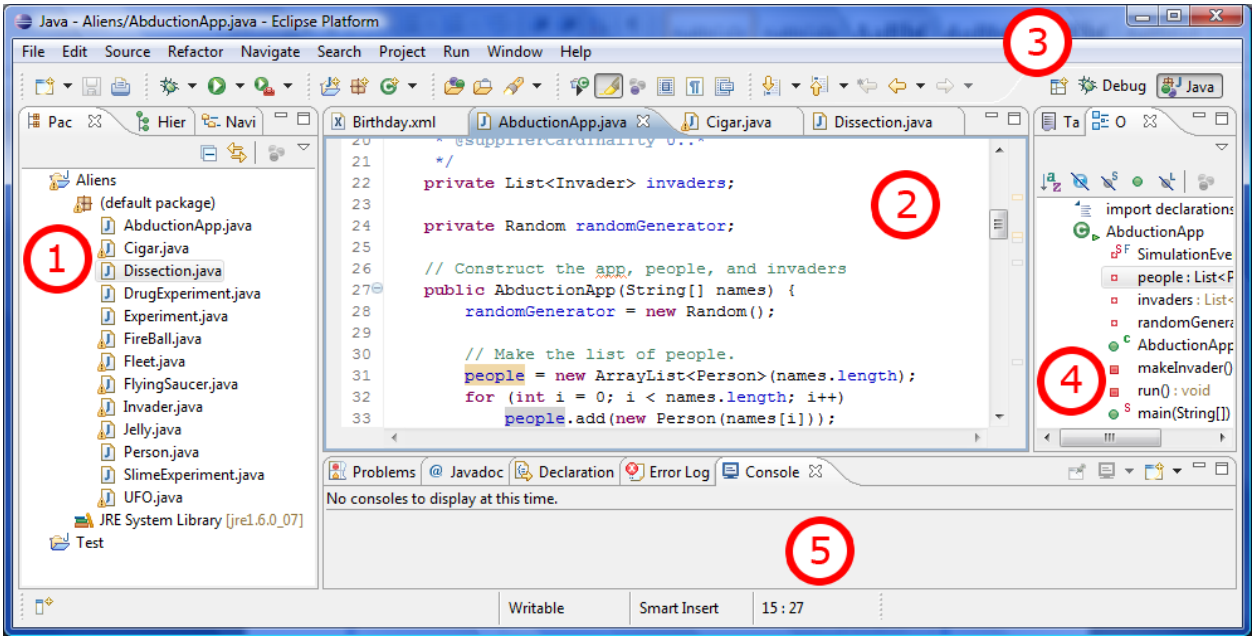


Figure 2.1: The Eclipse IDE

In figure 2.1 we present a screen shot of the default GUI provided by the Eclipse IDE. This figure shows a standard Java development environment. To the left of this image we can see a directory explorer (1) where source code files within the current projects may be accessed. The Java editor (2) is present in the center of the UI. We can also see a toolbar (3) that provides the ability to run code and access features such as debugging and refactoring. An overview (4) that provides fine grained detail of components of the current file can be seen to the right of the UI. At the bottom of the UI, a set of views (5) are provided. These views provide additional functionality such as a pane to view console input and output.

2.2 Visualising Software Metrics

Visualisation and software metrics are each large areas of research. In this report, we are concerned with the intersection of these elements: visualisations where the datasets are provided by software metrics. Various different types of these visualisations have been created by researchers. They include tree maps [34], city scapes [51], and force directed layouts [26]. Visualisations such a CodeCity rely on very simple metrics such as Lines Of Code (LOC) and Number Of Methods (NOM) [51]. However, attempts have been made to visualise more complicated and configurable product metrics [27].

In addition to these visualisations, there are a large number of modern tools that have the ability to calculate software metrics but do not have any method for visualising these methods, except for reports and potentially charts representing the collected data [3, 16, 26, 50]. This is important as it means that there are a range of tools that can provide the metrics data that we seek to visualise.

2.3 Augmenting Source Code

There is a variety of previous work that looks at the augmentation of source code. Some of this work is *in situ*, which implies that it must support the live editing of code as it resides in an editor. Other techniques, while they overlay data on source code, provide a read only display and cannot be considered *in situ*.

Of the integrated techniques that support *in situ* visualisation, the most commonly seen is syntax highlighting. This technique involves the alteration of the foreground text colour to indicate a property of the coloured syntax - it might be a variable name or a reserved language-specific keyword [46]. There has been very little academic work in this area and there do not appear to be any studies that provide evidence that this technique is beneficial. The earliest known usage of syntax highlighting is in the Live Parsing Editor (LEXX), a tool developed in 1985 which provided augmentations to a display of the Oxford English Dictionary [15].

One technique that has been suggested by Hendrix, Cross and Maghsoodloo is to overlay control structure diagrams directly over source code [22]. This research found that the there was a strong correlation with the existence of the overlay and improved productivity. Additional techniques, such as changes to background colour, icons in the margins and the underlining of source code can be found in some editors [46], but there does not appear to have been any attempt to provide a general purpose metrics visualisation framework. In addition, we have been unable to locate any academic work addressing their efficacy.

There are a few examples that augment static views of source code. The most well known is a tool called SeeSoft that was developed in 1992. A screen shot of that tool can be seen in figure 2.2.

This tool coloured individual lines of code in a read only display. The colouring represents process metrics that apply to the regions of the code displayed. SeeSoft is frequently cited in visualisation literature and has spurred a number of spin off visualisations. One example, the Visual Code Navigator, advances the metaphor by providing background colouring around nested code blocks and supports the abstraction of the source code level representation to a high level tree map for visualising large systems [34].

Within our research group, there has also been work in this field. In 2005 a tool named SeeSoftLike was developed. This tool is designed to provide an alternative to SeeSoft for modern OO languages. Additionally, it also supports the visualisation of product metrics [14].

We have been able to identify a number tools, like the tool we propose, that display software metrics information *in situ*. All of these tools have been implemented as Eclipse plugins. The EclipseMetrics plugin provides a table based report of the metrics it is able to calculate. It is also capable of providing simple margin icons to areas specifically referenced by the generated product metrics. In order for these margin icons to be generated, a static and user directed metrics generation process must occur [50].

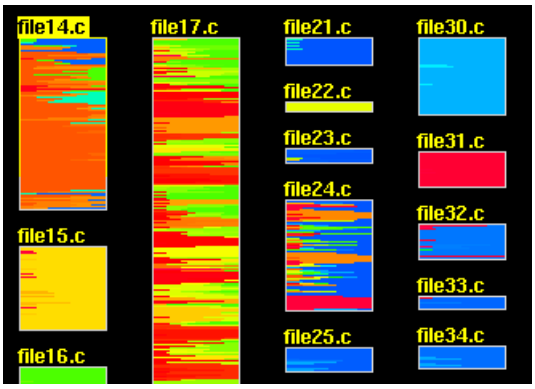


Figure 2.2: A screen shot of the SeeSoft tool

In the third year software engineering project at our university, the University of Canterbury, some groups have used code colouring to represent specific process metrics in the Java editor. There are also a proprietary tool, Clover, that provides a code coverage metric. This tool can exist as a plugin for Eclipse and provides basic augmentations in the form of background colouring within the Java editor [2].

These plugins provide very limited functionality and the provided functionality is in a contingent role. There appears to have been multiple independent occasions where the visualisation we seek to provide could have been beneficial and there does not appear to be any case where such a visualisation has been created as more than an inferior component.

2.4 Previous Publication

We have already submitted a conference paper on the material detailed in this report [20]. A copy of this paper can be found in appendix D.

3. *In Situ* Visualisation

In this section of the report, we examine the potential components of a visualisation that overlays software metrics data in the source code editor. Firstly, we outline the visualisation, then we attempt to provide a classification of the potential metrics that it might be desirable to represent. We then also classify the possible techniques for augmenting a source code editor, finally we look at the potential mapping and interpolation challenges between metrics and augmentations.

3.1 Visualisation Overview

This visualisation involves the placement of additional graphical elements into an existing source code editor, we term this variety of visualisation "*in situ*". These augmentations can consist of a variety of different additions, from code colouring to underlining sections of code. Each appearance of an additional element in the editor results from metrics data which has been mapped to that location. The development of this broad visualisation technique is our own work.

3.2 What Can We Measure?

In order to provide this visualisation, an overview of the different types of data we can measure is important. We can use this classification to inform the acceptable types of augmentation we can use and also to give an understanding of potential use case scenarios.

Essentially we can measure anything to which a quantitative value can assigned. In this research, we are primarily concerned with software metrics and heuristics. There are two main classifications of metrics; these are process metrics and product metrics. Process metrics are concerned with the development process behind the software engineering, whereas product metrics are an analysis of the actual code itself.

Process Metrics

Process metrics cover a wide range of types and have to potential to describe very useful facets of the program's development. These metrics tend to be related to the management and development practices used in software development. Traditionally, the gathering of these metrics has been automated [17, 26] into data retrieval from source code repositories or directly recording from a developer's actions in an Integrated Development Environment (IDE).

Source code repositories, and more specifically revision control systems, like Subversion (SVN) and Concurrent Versions System (CVS), allow certain types of process information to be gathered [17]. These systems keep track of the numerous and complex changes in software by recording the changes made in the system. This is usually achieved using a process of commits to a database.

The following are examples of some metrics that can be calculated with the data from these systems.

- Code Age - The relative date position of the last modification of a specific section of code. Other metrics allow the age of a specific file or the number of modifications to a line of code to be measured [17].

- Code Author - The last developer to work on a section of code. This may be extended, as with code age, to look at all of the different developers working on a section of code [17].

- Development Time - The amount of time spent writing and editing a section of code.

- Bug Tracking - The location and existence of system errors is also often incorporated into revision control systems.

It should be observed that due to the commit system used by repositories, users of these metrics need to be aware that the data obtained from these systems may not be fully up-to-date with modifications to an existing system. Care also needs to be taken to make sure that the difference between the number of commits and the date of the commits is considered.

In addition to using revision control systems, the monitoring of a developer directly through an IDE has to potential to allow further metrics to be collected. These have the potential to provide addition process data, that when combined with the metrics from a version control system, can give a more complete picture of the development process. Potentially, these systems allow fine grained knowledge of the workspace of each developer on the project. This allows the use of techniques like pair programming or refactoring to be monitored. It also allows a more complete understanding as to problem areas in the code by evaluating how a developer spends their time.

Product Metrics

Process metrics only provide half of the picture; a substantial range of metrics can be gathered by directly evaluating the created software product. This has two major components: source and runtime product metrics.

Source product metrics are those that are generated directly from source code. This generation is usually achieved using some form of parsing. In an optimal case, a parse tree that represents a model of the entire system is generated. From this tree, metrics data can be extracted. Some of the most well used metrics of this type are:

- Lines of Code (LOC) - The number of lines of code in a file or other section of code. This could include classes, methods and code blocks.

- Number of Methods (NOM) - The number of methods within a class.

- Cyclomatic Complexity - A measure of the number of independent paths with a system.

- CodeRank - An importance rating for a block of code. This system is based on Google's PageRank classification [37].

- Lack of Cohesion Of Methods (LCOM) - A group of metrics to measure the cohesion of methods.

- Fan In / Fan Out - The number of calls to and from a method.

In addition to the source types, runtime metrics are generated during the normal operation of the program. These can include such metrics as execution time, instruction path length and a variety of other metrics. These metrics are particularly useful during operations such as debugging.

Heuristics

A heuristic is measurable principal of some aspect of a software program. In the field of OO development, Riel presents a extensive list of heuristics designed to advise developers of best practices [40]. One example of these heuristics is to hide data within its class (Riel 2.1).

When a heuristic has be broken by a section of source code, a measurement can be generated. With regards to the example above, if a field in a class was made public, this would break the heuristic. In this visualisation we regards these heuristics as metrics in their own right as they provide measurable data.

Metrics Standardisation

One of the challenges that faces measurement, particularly in the case of software metrics, is that the measurements are not standardised. This creates problems as separate implementations of the same metric can produce radically different values [27, 33]. One common metric that has this problem is Cyclomatic Complexity.

In the case of our visualisation, this has the potential to be problematic as two reportedly identical metrics could produce different data and visualisation effects. This issue can be minimised by making sure that the type of algorithm used to calculate the metric is clearly specified and different measurements of the same metric are clearly labeled as such.

Metrics Data and Ranges

In order to provide an *in situ* visualisation, we need to provide a generalised form that can represent all of the metric types above. This can be defined by discovering what they all have in common and any specific features that need to be accounted for. Generalising metrics to a common model is important as it provides us with the ability to interpolate between metrics and corresponding augmentations; this is discussed in section 3.4.

We make a distinction between two concepts: a measurement and the metric it belongs to. A metric describes a particular type of measured value. In order to describe this, we define a metric in terms of three parts: a semantic understanding of what the metric means, a definition of the types of values that are acceptable, and a set of measurements that relate to that metric in the current context. For example, the Lines Of Code (LOC) metric can be semantically defined as the length of a document. The acceptable values for this metric are integer numbers between zero and positive infinity.

A measurement represents a particular instance of a piece of measured data. We model this by composing a measurement of two pieces of information: a value that is acceptable within the metric and the location of this measurement. A value could be a numeric value such as the number of LOC in a particular method or the name of developer who wrote that method. The location is a reference to that method.

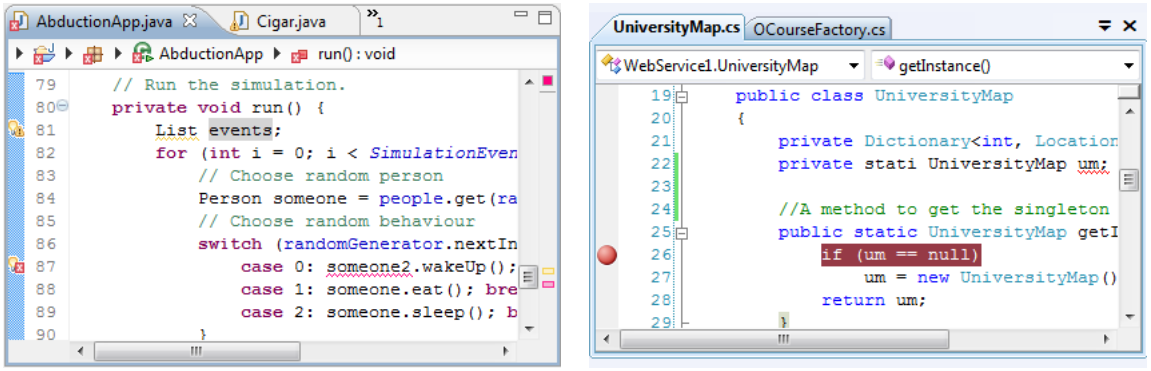
The LOC metric we described above has a set of acceptable values of $[0,\infty]$. We define a set of acceptable values as a range. This may be comprised of any number of different values and may

also be able to allow text as a form of acceptable value. For example, the name of a developer, "John Smith", would be an acceptable value within a certain range.

3.3 What Can We Visualise?

Display Regions

A source code editor is comprised of a number of elements; at the most basic level, this consists of a representation of a document that contains lines of text. Each line is displayed in sequence in the editor and in most cases, the concept of a wrapped line does not exist, the text continues for the entire length of the line. In order to navigate these documents, the most traditional method is a vertical scroll bar. In cases where the text is also too wide for the screen, a horizontal scroll bar may also be provided. The area of the document that a user can see is known as the view window. In addition to these features, source code editors may also provide internal and external margins on each side of the editor. An internal margin acts as padding from the outside border, whereas external margins provide additional panes around the editor. Both of these types of margin can be used to provide additional display information, for example they may note syntax errors in a document or the location of breakpoints [39]. Margins may also provide additional information such as a breadcrumb trail to help document navigation [46].



(a) An example of the Eclipse java source code editor

(b) An example of the Visual Studio C# editor

Figure 3.1: Two source code editors

Within these constraints, many source code editors provide additional functionality. This can include syntax highlighting, mouse over hovers to present additional information, and inline code completion techniques [46]. Context menus are also often used to provide features such as refactoring support and key combinations can allow document formatting options. Two examples of source code editors can be seen in figure 3.1.

Potential Techniques

There are potentially a wide range of augmentations that can be provided on top of a source code editor. We provide a classification framework to understand the potential types of augmentation

and the techniques that can be used. A basic example of the editor can be seen in figure 3.4(a).

Background Techniques

A background technique is any technique occurs behind the main visible content of the source code editor. This means that it occurs behind the written code. These techniques have the potential to be useful as they can provide information without excessively distracting or overloading the user with additional information. Possible techniques include:

- Colouring the background of a line or block of lines a colour. This colour represents an underlying metric quantity. See figure 3.4(b).
- Colouring a section of text, this may occur on a single line or continue over several lines. While this technique is similar to colouring an entire line, it only covers sections of a line where code is present. See figure 3.4(f).
- Vertical and horizontal colour gradients can be used to represent changes across a line, set of lines, or section of code. In these cases, fine grained metrics data is required to produce such gradients. See figures 3.4(c)(d)(e).

Foreground Techniques

A foreground technique is one that occurs above or at the same level as the main visible content. This area has been widely used to provide additional information to developers and so may have a more limited usage. Some examples of this additional usage are syntax colouring, underlining and boxing text, and also hover augmentations which provide additional information in the foreground.

These techniques have been widely used; this indicates that they will provide useful additions to this visualisation. As developers are familiar with this type of augmentation, care needs to be taken to avoid confusion with their use. This could be ensured by making sure that the colour schemes used by this visualisation do not overlap with existing techniques. Below are techniques that may be beneficial to the visualisation:

- Syntax colouring has the potential to be useful, however, due to its prevalence it is probably not wise to use this technique except in exceptional circumstances to avoid confusing the developer. See figure 3.4(g).
- Code styling has been used, however further potential exists. This includes changing the size, font and font styles (i.e. bold and italic) of a section of text [31]. In order to allow source code to be easier to understand, a non-proportional serif font type is nearly always used. The font size is also nearly always constant throughout the document.

  In order to keep the document clean and easy to understand, changing the font and font size within a document should be avoided as this leads to a common document presentation problem known as “ransom note typography”. However, changes between documents may provide useful information to a developer [25].
- Underlines and boxes could also be used, these could distinguish specific sections of texts. Different types of underline in different colours could be used. See figure 3.4(h). Apart from IDEs, this technique is common amongst text editors, such as representing grammar and spelling mistakes in Microsoft Word.

- Hovers also have the potential to provide useful information on top of elements when the user selects or holds the mouse over a particular location. This information could be as simple as textual information or more diverse information such as tables or other visualisations. See figure 3.4(j).

- Transparent overlays are an additional possibility, in this technique a screen showing an abstract representation is presented as a transparent layer over the source code. While this has the potential to be effective, it is liable to cause high levels of confusion and not be suitable for this kind of visualisation. See figure 3.4(i).

Peripheral Techniques

Peripheral techniques are here distinguished from foreground techniques as they occur in the margins of the editor. There are a number of possible techniques that are applicable:

- Icons and colour chips in either the internal or external margins are potential techniques for providing additional information. The colours and shape of the chosen glyphs can provide details to the user. The also provide a useful location for providing hover data. In terms of placement, the logical locations are the left and right margins and multiple chips could be stacked on each side of the editor. See figures 3.4(k)(l).

- Colour bars, like colour chips, can provide addition information. The main difference is that rather than corresponding to a single line, colour bars can provide an indicator over a multi-line section of code. See figure 3.4(l).

Ambient Techniques

In addition to the three types of techniques listed above, ambient techniques can also be used. These techniques are often not immediately apparent to the user and would serve subtly indicate features of the source code. In this project it is not our intention to develop these further, however, their existence should be considered as they are applicable without the user changing their focus. Here are a list of some that may be useful to developers:

- Auditory techniques, such as using sound to provide feedback and information content may be useful [7]. This technique would allow the developer to judge a piece of code by its relation to an audio signal with characteristic qualities.

- Olfactory techniques may also be useful to the developer. A characteristic smell could define a positive or negative quality in a code. This could take the form of a literal code smell [53].

- Outside the immediate area of focus, visual techniques could provide indicators to the user. These could either be in the surrounding area of the screen or external to the main display. They could consists of changes in colour and light intensity. Providing definitive glyphs is unlikely to be productive as they would distract the user from their current focus.

- Additionally, changes in temperature and vibration could also provide hints to the user [41].

HCI Concerns

While all of the types of augmentation listed above have the potential to be displayed in or around a source code editor, there are reasons why certain versions of these augmentations may not be desired or why combinations of these augmentations may create undesirable effects. These HCI issues are discussed below.

Information Overload

In a case where a developer is presented with to much information to concentrate on they will find in hard to be productive. This will also be annoying to a developer and they will stop using the visualisation. This should be avoided by providing the developer with tools to customise the augmentations that are visible.

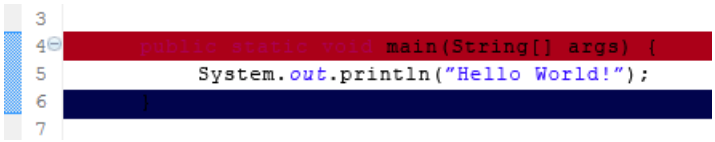


Figure 3.2: An example of poor contrast using background colouring

Contrast Issues

The choice of the shape and colour of elements to display is important. These features should be intentionally designed so they are easy to distinguish from existing elements in the source code editor and other visualisations. A particular concern is that the choice of colours to present an augmentation may provide an unpleasant combination with other augmentations or may be similar to the syntax colouring in the document. This would make distinguishing text from an augmentation difficult, especially if the augmentation was layered directly above or below text, this can be seen in figure 3.2. Contrast is an important issue and one that users need to be informed about.[1]

Creating Inferences

When multiple augmentations are displayed within the source code editor at any time, the user has the ability to use the combined pieces of information to form inferences about the underlying data set. Making these inferences is a huge benefit to this type of visualisation and should be encouraged. This means that different types of augmentations need to work along side each other and that the same metric needs to have the potential to be shown by multiple augmentations.

Attention Seeking

A range of possible augmentations exist. The developer should be aware of their existence and be able to refer to them with ease. However, except in very specific circumstances, they should avoid

[1] It is worthwhile noting that a significant proportion of software developers are colour blind, therefore visualisation techniques that do not exclusively make use of colour should be considered.

trying to seek the developers attention. This means that motion or quickly changing scenes within the user's area of focus should be avoided.

3.4 Mapping & Interpolating Results

We have identified two elements: a software metric with a range and an augmentation with adjustable properties. In order to make use of them, we need to define a way of mapping between the two elements to ensure that the information within the data is preserved. In order for this operation to be completed, we propose three stages:

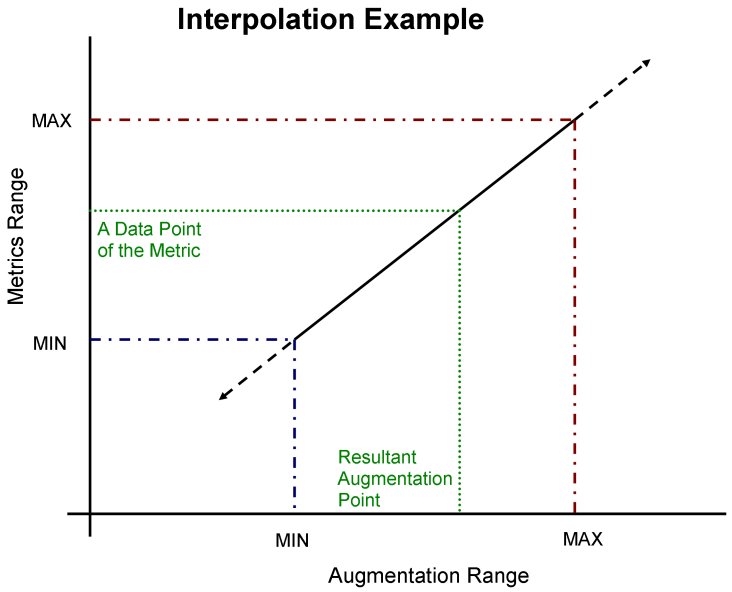


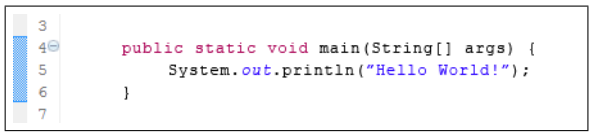
Figure 3.3: Linear interpolation

1. Each instance of metrics data is checked against the range of that metric to make sure that the data point provided is valid.

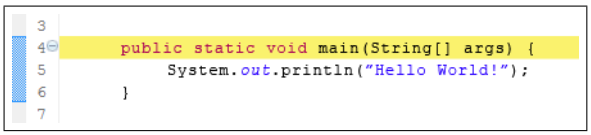
2. A range is used for the desired augmentation. This range consists of a set of possible values that can be displayed. These values may correspond to different colours, shapes or images.

3. After both of these initial conditions have been established, a mapping between the two data sets can take place. In this mapping, an interpolation takes place between the range of possible values established for the augmentation and the those established for the metric. A simple interpolation is shown in figure 3.3

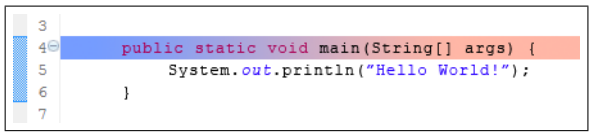
In order to make sure that this mapping is smooth and valid across all points on the real number spectrum (including values that tend to positive or negative infinity), different types of interpolation can be used, these can be based on an exponential function, a logarithmic function or a trigonometric function (tanh).



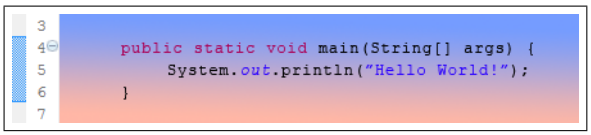
(a) The editor without any augmentations



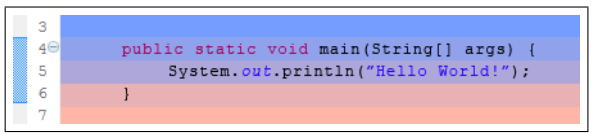
(b) Background code colouring



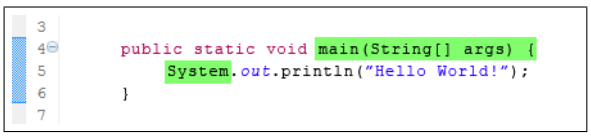
(c) Background code colouring with a horizontal gradient



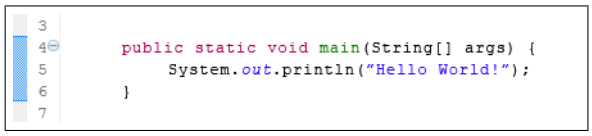
(d) Background code colouring with a vertical gradient



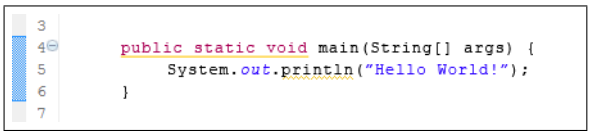
(e) Background code colouring with a line size step vertical gradient



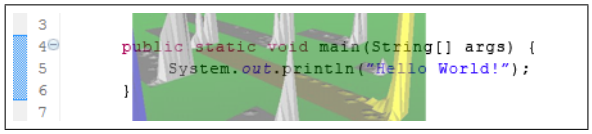
(f) Background code colouring of a fine grained set of characters



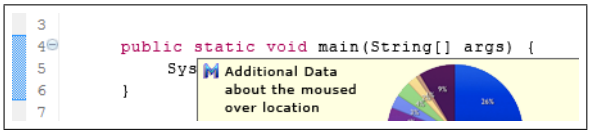
(g) Additional syntax colouring on the method name "main"



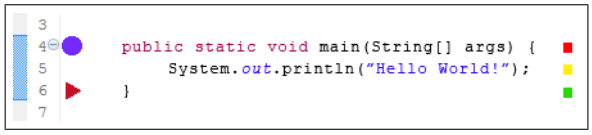
(h) Two different types of underlines, a single straight line and a swiggle



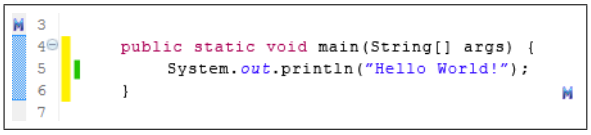
(i) Other software visualisations can be used to directly provide overlaid content



(j) A text hover can provide extra information if a location is moused over or selected



(k) Colour chips in the left and right margins, note that different shapes and colours can be used to provide information



(l) Colour bars and icons can be used to provide peripheral information

Figure 3.4: Examples of possible augmentations within the source code editor

4. CoderChrome Design

In this chapter we will detail and discuss the tool we have produced. In addition to providing an overview, we will also detail its integration into an IDE, its architecture, further features it provides, and the limitations of the system. At the end of this section we also discuss the extensibility of the tool.

4.1 Tool Overview

We have created a tool, CoderChrome, that realises the visualisation technique described in chapter 3.[1,2] This tool is implemented as an extensible Eclipse plugin which integrates with the Java editor this IDE provides.

Figure 4.1 shows a screen shot of CoderChrome in action. This tool allows multiple augmentations, that are mapped from existing or generated metrics, to be displayed in Eclipse's Java source code editor. These augmentations allow metrics information to be presented in a variety of ways.

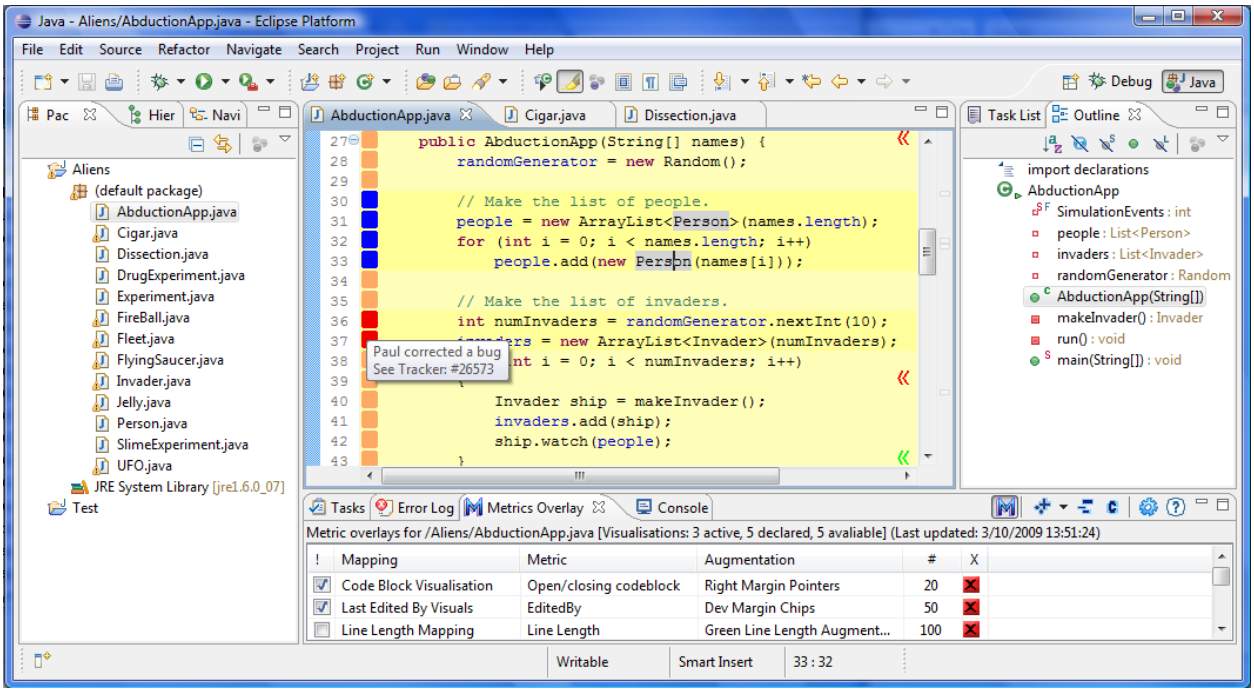


Figure 4.1: CoderChrome in use in Eclipse's Java editor, a combination of augmentations is shown

In order to provide discrete user control over the tool, two additional components are also

[1] Thank you to Neville Churcher for the suggestion of this name.
[2] The tool and its user documentation is available on request to the author

provided. The first of these is a view, this allows users to select a set of mappings to investigate and control which of them are active. This view is described in greater detail in appendix A. In addition to providing user control, this panel also facilitates access to help and preference controls. A screen shot of this view, populated with five mappings, is shown in figure 4.2.

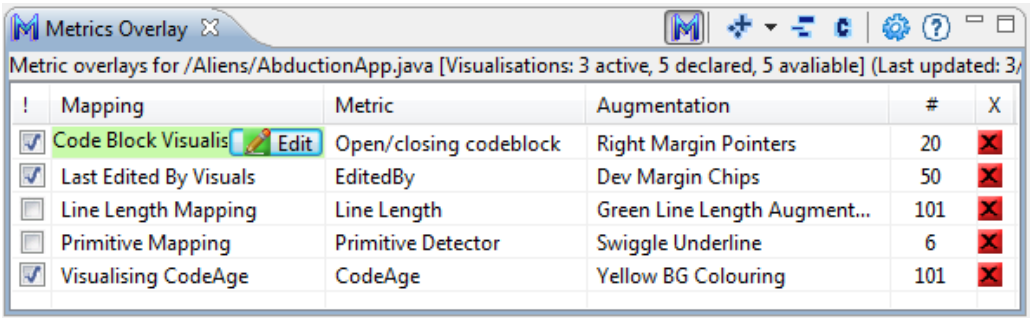


Figure 4.2: The view provided to control the current mappings

The preference pages allows the user very extensive and modular control over the mappings, metrics and augmentations in use by the system. This preference system is facilitated by the very flexible model which is described in section 4.3. Each of the preference pages are shown in appendix B. The amount of control that these two components provide is a significant benefit of the system.

4.2 Eclipse

In this section we discuss our decision to use implement our solution in the Eclipse IDE and provide information about the specific components of Eclipse that we have made use of. Figure 2.1 shows screen shot of the default Eclipse configuration. Eclipse is an industrial strength IDE that allows the implementation of functionality using a plugin architecture, as such, it is designed from the ground up to be extensible.

Some of the benefits that Eclipse provides are:

- Usage - Eclipse has the second highest IDE market share after Visual Studio. It also has a 70% usage among the Java development community [4,36]. This makes developing for Eclipse attractive as there is an increased potential of reaching a wide audience.
- Development Community - Eclipse has a strong and active development community, meaning the existing problems can be quickly solved [5].
- Free - Eclipse is free to its users. This means that deploying applications for this IDE is cost effective [1].
- Open Source - Eclipse is a completely open source platform. This means that creating new components for Eclipse is made easier as developers have access to underlying source code of the objects they are extending or using [1].
- Plugin Infrastructure - Eclipse is a completely modular IDE that successfully uses the concepts of extensions to provide added functionality.

The Eclipse IDE provides developers with integrated support for developing large software applications. This support includes integrated debugging, synchronisation with source code repositories, a project explorer, and of course, a source code editor. The provided GUI is highly modular, providing support for developers to dramatically change the layout of their workspace as desired.

Some of the central components of the Eclipse infrastructure that are important to this tool are:

- Java Development Tools (JDT) - This set of modules provides Java related functionality into Eclipse. This includes:
  - The ability to directly run Java applications with console input and output.
  - Java specific navigation to explore source code directories and open source code files.
  - An editor to allow code to be written and modified. This editor features a variety of useful functionality including tabbed documents, breadcrumb trails, break point handling, syntax highlighting, and code completion.
  - Further functionality, such as debugging support and integrated unit testing.
- Simple Widget Toolkit (SWT) - This is a Eclipse specific graphics library for creating User Interfaces (UI) and underlies the UI that provides the graphical content for Eclipse.
- Rich Client Platform (RCP) - This is a set of modules that are designed to facilitate the building and deployment of components within the Eclipse platform.

4.3 Architecture

The system is comprised of three components that can be seen in figure 4.3. The `EditorListener` component aggregates changes to the existing Java editor into a usable format, the `MetricsDataProviders` use this data to add metrics information, and the `MetricsOverlay` component keeps track of the current model and user preferences. It is also responsible for supplying information to the editor. These components are examined in greater detail in the following sections.

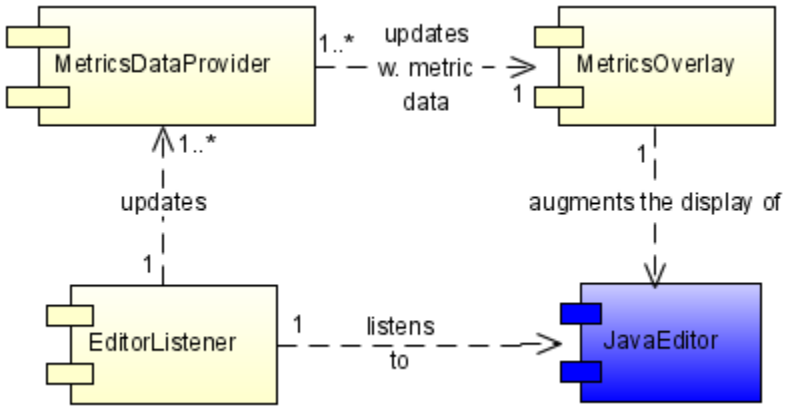


Figure 4.3: A component level architecture of CoderChrome

While designing this architecture, a key design consideration was the use of the "Tell, Don't Ask" maxim [45]. None of the components request information from those earlier in the chain, they wait until the information is provided. For example, the `MetricsOverlay` component keeps its current set of metrics data until a new set is provided. This decision is important as it produces a design that is architecturally clean (due to low coupling) and easy to maintain.

A central concept in use in this component model is the Observer design pattern. [19] This pattern allows us to abstract the coupling between the components we use. It also allows us to provide a broadcast communication model and the ability to provide updates at any time.[3]

Listening to the Editor

The `EditorListener` component mentioned earlier provides the ability to monitor changes made to the current editor the user is working on. A wide range of changes can be made by the user and these changes are represented using a complex set of event generators. The `EditorListener` component has the following behaviour:

1. Various listener objects are created to make sure all valid events are caught for the desired document types.
2. Events are captured as they occur.
3. Captured events are checked for validity.
4. Addition details on the registered events are sought to ensure that a consistent set of data is available for all events.
5. The captured event is translated into a simplified form that generalises the changes to make later computation more straightforward. The potential generalisations include: newly opened documents, modified documents, and changes in focus between two already open documents.
6. Using the Observer design pattern, an observable interface then provides the data about the updated editor to any listening `MetricsDataProvider`s.

This component is designed to be extensible, so that different file types within different Eclipse editors can be monitored as desired.

Metrics Data Providers

In the most basic case, a `MetricsDataProvider` provides the `MetricsOverlay` component with metrics data. The default behaviour of such a component is to observe (using the Observer design pattern) the `EditorListener` and on an update, generate a new set of metrics data as an instance of the system model (see section 4.3), and provide it to the `MetricsOverlay` through an Observer based API. However, `MetricsDataProvider`s can provide more complex behaviour to suit a variety of purposes; these features are detailed below.

- Multiple providers can be created and used concurrently. Each provider can contain one or more types of metric. The `MetricsOverlay` component handles the data sets provided by multiple sources transparently.

[3] Further information on this design pattern can be found at http://wiki3.cosc.canterbury.ac.nz/index.php/Design_patterns

- Not only can metrics data be provided, but these providers can also provide default or recommended sets of mappings and augmentations to view them with (discussed in section 4.3).

- While the `EditorListener` component provides an easy source of updates for the system, developers can choose to implement their own set of listeners and update the model as they see fit. The `MetricsOverlay` component attempts to manage all provided datasets and maintain their validity as far as possible. This is valuable as existing metrics generation tools can provide metrics data without having to interface with the `EditorListener` component.

- This flexible architecture style provides the ability to deal with both static and dynamically generated metrics. In this research we focus on dynamic metrics as they more clearly fit with the usage of a source code editor, and by implication fit the *in situ* paradigm more closely.

- While `MetricsDataProvider`s can provide data through an API, an XML representation can also be provided to the `MetricsOverlay`. This approach provides flexibility when integrating existing XML based tool sets and reduces the difficulty of providing valid data. In this case, an XSLT transform is all that is needed to transform data into an appropriate form. Within our research group, the use of XML for this purpose is common and will allow us to integrate our existing tools [26].

System Model

The system model provides a straightforward and extensible model of the visualisation. It consists of the main concepts and relationships within the visualisation. A simplified UML class diagram can be seen in figure 4.4. In this model, we can see the concept of a `Mapping`. A `Mapping` holds a `Metric` and an `Augmentation`, it is responsible for interpolating between them. This is achieved using the `InterpolationStrategy`, which is an implementation of a Strategy design pattern [19]. A `Metric` holds a `Range` that describes the `Metric`, and a set of `MetricSections` that represent the calculated measurements for a particular `Metric`. An `Augmentation` of a specific type represents an overlay in the editor. The `AugmentationStrategy` is in charge of implementing this overlay.

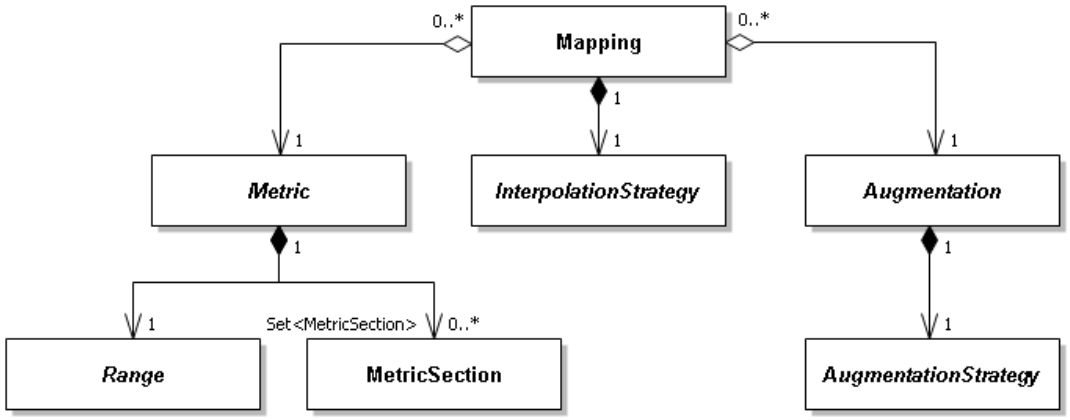


Figure 4.4: The system model

In order to allow us to translate from a measured value of a `Metric` to a displayable `Augmentation` value, we use an `InterpolationStrategy`. The subclasses of `InterpolationStrategy` implement various types of interpolation. They require a minimum and maximum for the `Metric`'s range and for the `Augmentation`'s range. When a measurement is provided for a specific `Metric`, an interpolated value is calculated. A variety of different types of `InterpolationStrategy` are provided, including logarithmic and exponential based interpolations. The equation of a linear interpolation is shown in equation 4.1; 'A' refers to the `Augmentation` values and 'M' to the `Metric` values.

$$\delta = \frac{A_{max} - A_{min}}{M_{max} - M_{min}} \qquad \phi = A_{max} - \delta \cdot M_{max} \qquad A_{val} = \delta + \phi \cdot M_{val} \tag{4.1}$$

An important characteristic of the system model is that is allows interchangeable components. `Metric`s and `Augmentation`s are interchangeable and can be reused across multiple `Mapping`s. In addition, the classes within the system have been designed to be self describing in terms the user can understand. Interfaces ensure that this functionality is consistent across the model.

Metric

In the previous figure (4.4) a simplified diagram of the system model was presented. In figure 4.5, a fuller representation of the `Metric` hierarchy is provided. A `Metric` represents a definition of a particular type of measurement of software. A `Range` defines the possible values that of a measurement. `Range`s are described in the subsequent section. A `MetricSection` represents a specific measurement; essentially this consists of a value and a location in the current document. A location is a region in a Java file.

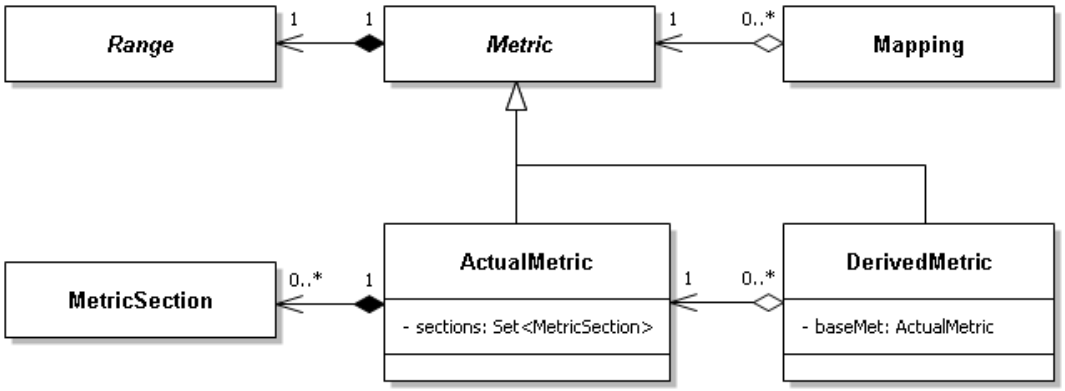


Figure 4.5: The metric components of the model

There is an additional level of complexity in this implementation. A key concern was to avoid the duplication of the measurement datasets that these `Metric`s represent. However, we also wanted to be able to manipulate a `Range` to create different effects and maintain reusability. In order to do this, we use a modified Composite design pattern [19]. An `ActualMetric` is the only class of `Metric` that can actually contain a set of `MetricSection`s. But, these `MetricSection`s can be reused by creating a `DerivedMetric` representing an `ActualMetric`, but holding a completely new `Range` object.

Range

A range determines the minimum and maximum allowable values of a set of data. Data values that fall outside of this predefined range are discarded. In addition, we can refine a range as ordinal, nominal, interval or ratio scale. Ordinal numbers represent a ranking; first, second, third etc. Nominal assigns a specific name to a value, one example in software metrics is the name of the last developer of a section of code. An interval seeks to define a set of acceptable values on the real number line and ratio describes a measurement as value and a scalar modifier.

Additionally, a range can consist of one or more sub-intervals. For example, when investigating Lines Of Code (LOC) we may only be interested in method bodies of 1-3 lines and greater than 15 lines in length. These two sub intervals need to be described in a model of a range. A final issue is that of using a sub interval to describe a discrete point on a data set. With our LOC example, we may wish to classify long and short method types separately.

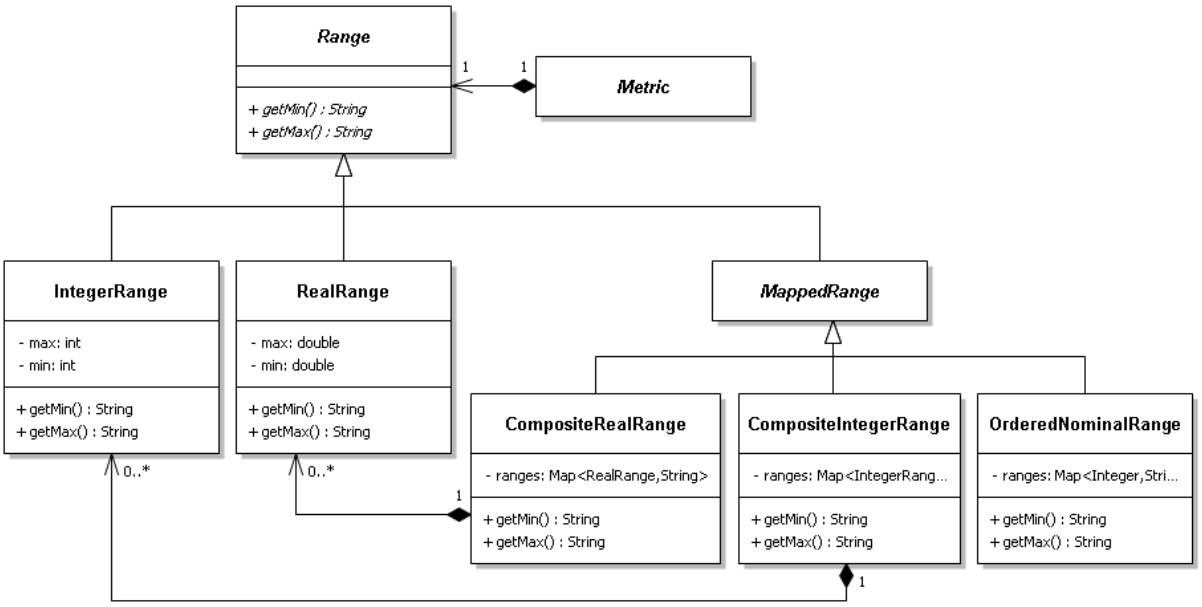


Figure 4.6: The range components of the model

The model we have developed generalises these concepts and provides a reusable code base for describing ranges. A UML diagram of `Range` and its subclasses can be seen in figure 4.6, in this case only some of the methods and variables are shown. For general usage, two simple range types have been provided - `IntegerRange` and `RealRange`. These classes allow for ranges with a simple maximum and minimum value. The `IntegerRange` class is restricted to integer values, whereas the `RealRange` can model any floating point numbers, including positive and negative infinity. `OrderedNominalRange` provides the ability to use nominal data sets. In order to maintain appropriate mappings, this set requires an ordering which is provided by assigned integer values to each nominal point. `CompositeIntegerRange` and `CompositeRealRange` use the `IntegerRange` and `RealRange` classes respectively to provide the ability to have a series of disjoint sub intervals in the data set. These objects provide the ability to calculate the minimum and maximum points

of the range and also to provide normalised data for the program's internal operation.

While this model provides sufficient descriptive power to provide a range for a majority of cases, there exist some cases that this model cannot describe. These cases are those where a differentiation is needed between sub intervals, specifically whether they represent discrete or ranged values.

Augmentation

An augmentation is any element that can be displayed within the source code editor. These augmentations directly relate to those described in section 3.3. In our implementation, we provide an `Augmentation` hierarchy that represents the types of augmentation that can be displayed by the editor. `Augmentation` objects know the type of element they are; however, they do not have the ability to update themselves. This functionality is provided by a parallel hierarchy of `AugmentationStrategy` and its subclasses. A much simplified diagram of this relationship can be seen in figure 4.7.

The decision to separate these two elements breaks Riel's heuristic (2.9) to keep related data and behaviour together [40]; however, in return it offers a more flexible implementation that allows the implementation of an AugmentationStrategy to change with the addition of a new type of editor in the system without having to add a new Augmentation. It also simplifies the design by abstracting apart these complex implementations.

We have implemented a variety of types of augmentation described in section 3.3. These augmentations include underlines, background colouring and margin chips and are detailed more throughly in our results section (5.2).

Within the types we have implemented, we have complete control of the colour schemes. These colour schemes are controlled by a set of colour interpolators, the models of which are discussed in section 4.4.

It should be noted that the developers of the JDT and the Java editor hid some of the functionality required to make direct changes to certain elements of the displayed editor. In order to provide this functionality, we make use of Java's reflection library to provide access to these components. We believe that the reason this component was hidden is that modifying this component often has unexpected consequences, such as causing the entire editor to freeze or incorrectly display information. As such, when adding a new augmentation, a significant amount of testing must take place to check that these conditions are avoided.

4.4 Additional Functionality

Plugin Infrastructure

Eclipse provides a plugin architecture using a module called the Rich Client Platform (RCP), this provides the functionality to provide extensions to Eclipse. CoderChrome consists of a number of plugins that can be deployed within the Eclipse environment. Earlier we showed our component based architecture in figure 4.3, the `EditorListener` and `MetricsOverlay` components each correspond to a single plugin in the Eclipse architecture. Each different type of the `MetricsDataProvider` component also corresponds to a single plugin (although the implementation of these may require addition plugins). The plugin architecture therefore strongly models our underlying component architecture.

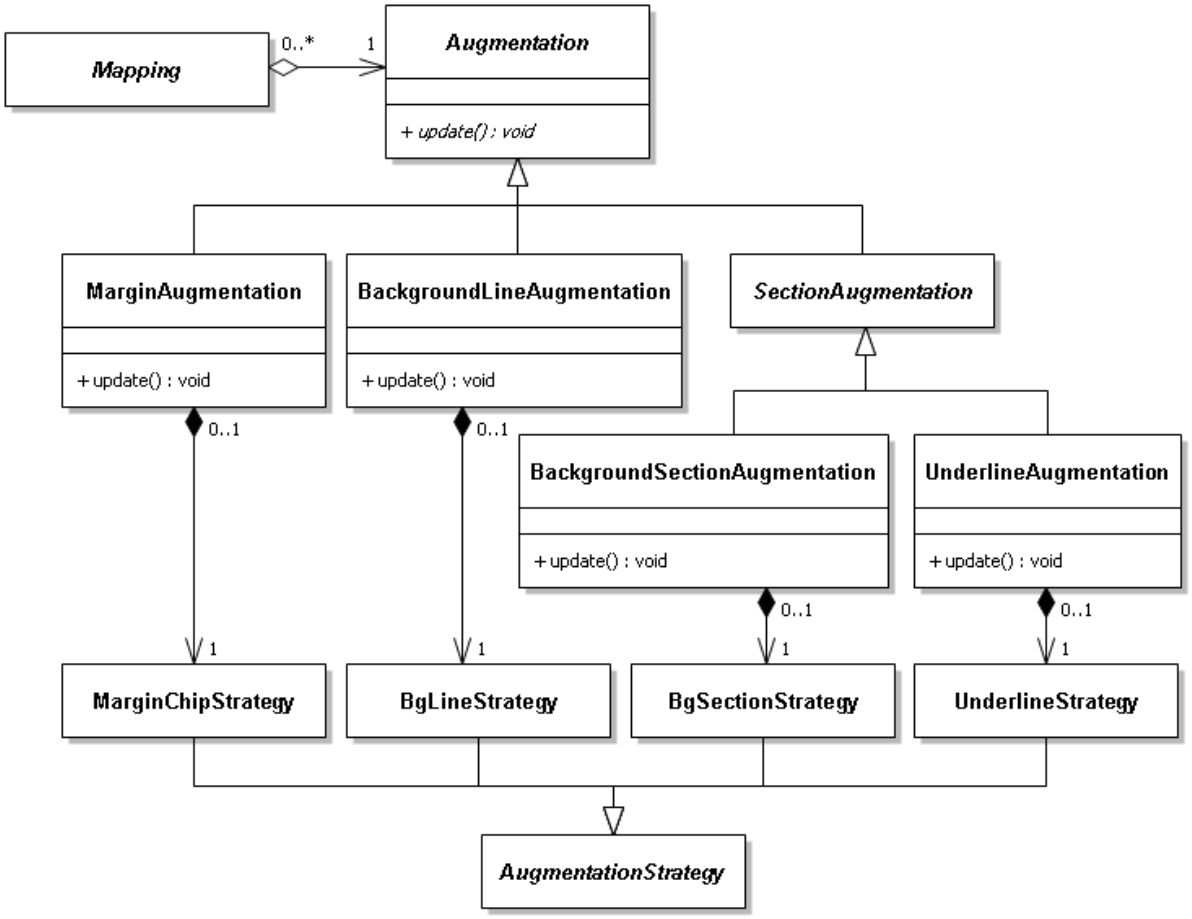


Figure 4.7: The augmentation components of the model

In order to provide interaction between these plugins and the rest of Eclipse's infrastructure, Eclipse implements a concept of extension points. Plugins can use provided extension points to add additional functionality to underlying components. Conversely, plugins can also provide their own extension points that can be implemented by other plugins. Implementing and using extension points is achieved with a complex set of XML control documents that govern the behaviour of each extension point. Once these extension points have been declared, the functionality they provide can be used by programmatically instantiating or extending classes in the Eclipse API.

CoderChrome extends a number of existing extension points. These extension points allow CoderChrome to provide functionality such as integrated help, preference pages, a 'view' and changes to the editor. These extension points also provide lower level controls such as the icon buttons in the `MetricsOverlay` view.

CoderChrome implements its own extension points. These provide a formalised set of controls over the underlying Observer based architecture which we discussed earlier. `EditorListener` and `MetricsOverlay` both implement extension points that have to be utilised by any created `MetricsDataProvider` plugins.

Data Persistence

When `MetricsOverlay` component receives an update from a `MetricDataProvider`, some actions have to be taken to integrate this data with the current model. This integration is important as it provides a persistent interface and set of functionality for the user. In order for this to occur, the following steps take place:

1. A new model is received from a `MetricDataProvider`. If the received content is in XML, a new system model has to be created based on its content.

2. The supplied model is merged with the global system model. In order to do this, the names of elements within the model, user preferences and user created content need to be validated. Depending on the type of element (`Metric`, `Mapping` or `Augmentation`), if there is a conflict, the system decides to either discard a copy or merge two elements.

3. The updated global model is stored.

4. The source code editor is updated to display any changed content.

Colour Models

One of the main elements we seek to interpolate across is colour. In order to do this a variety of techniques may be used, each of these techniques relate to a different colour model. By choosing different colour models we gain an interpolation that has certain features. Within the current development, we have concentrated on the providing the commonly used RGB and HSV colour spaces. We discuss a broader range of models below with reference to their qualities for interpolation and computational complexity.

- RGB (Red Green Blue) - This is the standard additive colour model that is employed to display colour on a screen. In the case of LCD screens, this is made possible by the combination of light from three sub-pixels - one red, one green and one blue. It is important to recognise that this colour model is device dependent. In our implementation, we use a standard linear interpolation between the two points represented as 24bit colour values. While this model is useful and computationally cheap, the interpolation it provides does not appear natural, as interpolated colours tend to have a tendency to distort ranges that include grayscale values [44].

- HSV/HSL (Hue Saturation Value and Hue Saturation Lightness) - Rather than providing an additive or subtractive colour space where the elements represent a specific colour which are effectively blended to produce a final result, these colour spaces abstract out the concepts of hue and saturation from this model, making them more controllable. This results in a cleaner and easier to comprehend linear interpolation. It has the disadvantage that in order to convert from the default RGB values, a transformation is needed, this transformation is potentially computationally expensive for documents utilising many colour transformations [28, 35].

- CIELux/CIELab (Commission internationale de l'éclairage - 1931) - These colour spaces are designed to model the visual space accessible to normal human vision. It potentially provides the most accurate method of interpolating between colour points; however, the calculations

required are more complex and more computationally expensive, making this model less attractive. Additionally, the colour spaces these components describe are unintuitive [18].

- sRGB (Standard RGB) - This RGB colour space is derived from a subsection of the CIE XYZ colour space. It is designed to provide a device independent colour space, however it does not provide a significant advantage of an RGB colour space for the purposes of interpolation [47].

- CMYK (Cyan Magenta Yellow blacK) - This subtractive colour space is primarily designed for printed documents. While its direct use in this application is limited, it has the potential to be useful in cases where printed views of a document provide additional benefits. This could possibly include code reviews of printed source code [18].

4.5 Extensibility

One of the primary goals of the system was to create an extensible model that can be used for a variety of purposes. Below, we describe the different purposes that we anticipate our design being applicable to.

Within Eclipse

The system is designed to be easily extensible to different editors and perspectives within the Eclipse framework. This can be achieved by extending `AugmentationStrategy`, within the system model, to implement the augmentations for the new editor and specifying a different type of file to listen to in `EditorListener` component. This allows the visualisation to be portable to different languages and specially language implementations in Eclipse. For example, this would allow a C++ editor or a PHP editor to display these visualisations.

Using this extensibility, new language implementations can use this model to provide syntax colouring implementations, in other words, this *in situ* visualisation tool could supersede existing syntax highlighting systems by providing the ability to swap between different types of augmentation.

Across IDEs

While the `EditorListener` component and the provided `Augmentation`s are specifically designed for Eclipse, the architectural structure and the design of the system model are potentially applicable to deploying this visualisation technique across other IDEs. This is a realistic goal as many other IDEs have similar plugin architectures to Eclipse.

Across Visualisations

While CoderChrome is specifically designed to apply augmentations directly on top of source code, this system could be extended to augment other components within an IDE environment. For example, project explorers are designed to provide a hierarchical representation of classes and packages/namespaces within an application, these entries in the explorer could be augmented with metrics data to provide developers with a higher level augmented view of software that remains within the direct view of the developer.

Another example would be to use the developed framework to provide augmentations to a UML diagramming tool, preferably one that provided round-trip diagram and code generation. This would also allow developers to see metrics applied to more abstract representations and allow users to drill down and explore an existing system more throughly. This might allow developers to be informed of design patterns in use within the system or if a developer has broken one of Riel's heuristics regarding inheritance [40].

In order to provide this functionality within Eclipse, much of the existing infrastructure would remain. This would only require the addition of new functionality within the `EditorListener` component and the development of new types of `Augmentation`. This would also require an extension to the `MetricSection` component to provide a more appropriate description of the location of a new overlay.

5. Results & Evaluation

In this chapter, we show our results and report on our progress in evaluating the effectiveness of our tool, CoderChrome. Meaningful evaluation of a tool such as this is a challenging task. Ideally it would involve a long term study in a real world setting. This section of the report includes a discussion of our plans in this area.

5.1 Results and Evaluation Discussion

This project is part of a larger software engineering research programme that aims to improve software metrics and visualisations, particularly as they are applied in industrial development environments. Evaluation of the effectiveness of metrics and visualisations is an important part of this larger programme [8, 52], but one that becomes possible only once industrial strength metrics and visualisation tools have been deployed.

CoderChrome has been constructed to be robust and extensible in order to support future research, particularly in commercial environments. However, in order to answer questions about the value of metrics and visualisations to software engineers, robust tools to acquire metrics data in an Eclipse environment must first be developed. Once this has been done and the metrics supplied to CoderChrome, it will be possible to test the effectiveness of *in situ* visualisation for software engineering purposes.

In this context, the primary goal of this project is to enable future research by providing a tool that supports experimentation. The principle question we need to answer comes from a software engineering standpoint: is it possible to extend a widely used IDE to support *in situ* visualisation? The answer is yes, but it is not easy. We report on more specific results in section 5.2.

Although it premature to attempt full scale evaluation of the usefulness of *in situ* visualisation, we conducted an informal evaluation to identify the most significant issues with using CoderChrome. This evaluation is described in section 5.3.

5.2 Results

We have achieved our goal of providing *in situ* visualisation within an industrial strength IDE. We have succeeded in realising the visualisation techniques that we proposed in section 3.

One of the required features of our tool was that the augmentations it provides can be turned off so that Eclipse development environment appears in its normal form (an example was shown in figure 2.1). This requirement has been met; when all augmentation is turned off, the editor displays its default behaviour, including features such as syntax highlighting.

The requirement that the tool work in an industrial software development setting gives rise to several more specific requirements:

- The tool must integrate seamlessly with the Eclipse environment as a plugin. This has been achieved; a screen shot of the plugin in action was shown in figure 4.1.

- Part of making the tool acceptable in real world software development environments is the requirement that it be highly configurable. Figure 4.2 provided a screen shot of the configuration panel, which allows the current visualisation to be customised. Appendix A provides further details of the capabilities of the panel.

- Interoperability with other Eclipse plugins is an important concern. We have tested CoderChrome in a realistic Eclipse configuration containing a variety of common plugins in order to demonstrate that it coexists without problems.

- CoderChrome does not introduce significant performance overhead to developers when using Eclipse. We have tested CoderChrome on a number of realistic systems and have been unable to detect any noticeable impact on performance.

- Full user documentation is provided.

CoderChrome successfully implements the family of visualisations proposed in section 3, including peripheral augmentations, foreground augmentations and background augmentations. Foreground augmentations can be seen figure 5.1(b), background augmentations can be seen in figures 5.1(a)(d), and peripheral augmentations can be seen in figures 5.1(b)(c)(d). The proposed ambient visualisations were not attempted as they are more experimental and require specialist hardware.

There have been a few limitations with the types of augmentation we are able to portray. We have not provided vertical margin bars, gradient background colouring or provide complex types of hover. Additional development and refinement of CoderChrome will look into providing these types of augmentation.

5.3 Informal Evaluation

In this informal evaluation, two groups of five developers each were asked to install CoderChrome and use it on a substantial team development project. The developers were third year undergraduate computer science students who were familiar with Eclipse and the plugin development platform (RCP). We did not place restrictions on our users as we did not want to constrain the feedback we received.

The feedback we received about CoderChrome was revealing. There was a general agreement that developers "liked the visualization options provided". With regards to using the plugin to implement their own metrics, they commented on the appropriateness of the system design and the ease with which the system can be extended. The ability to programmatically set the system's behaviour was received favourably.

Other positive feedback we received was that:

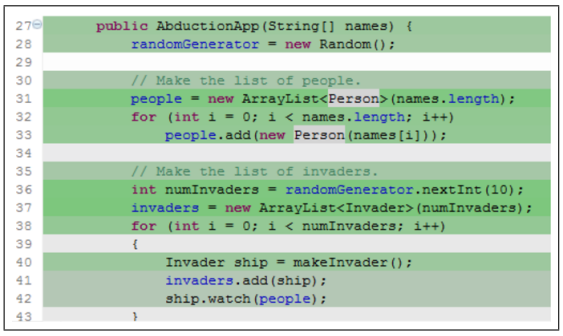
- The system is unobtrusive to the user, one comment was that it "fits neatly into the Eclipse environment".

- It has the ability to set preferences to appreciated.

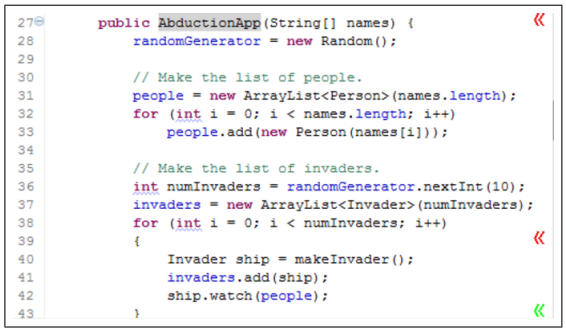
- The help provided to the user was sufficient.

- The system is easy to install.

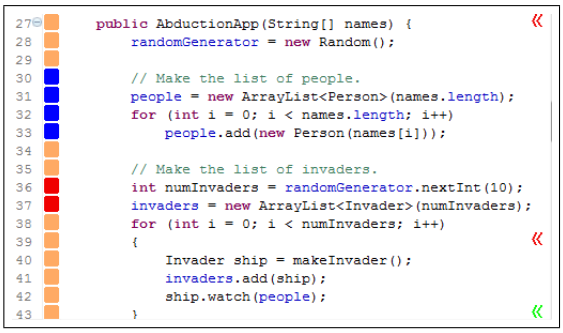
Two critic



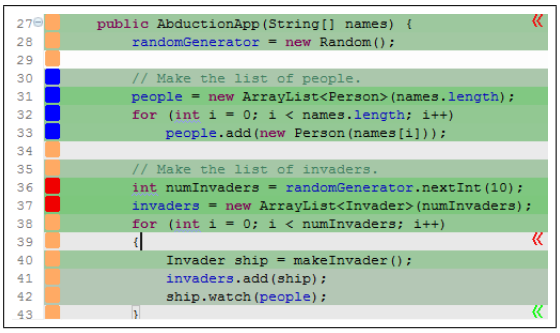
(a) Background colouring of a line length metric



(b) Underlined primitives and colour chips representing code blocks are subtle augmentations



(c) Colour chips indicating the author and code blocks



(d) Multiple augmentations in use

Figure 5.1: Screen shots of CoderChrome augmenting in the Java editor

6. Discussion

In this section we explore several topics related to the visualisation design and implementation we have produced. We discuss the different requirements of developers and managers and the implications this has for our tool. We then look in more detail at the consequences of using *in situ* visualisation for development.

6.1 The Roles of Developers and Managers

Traditionally, software visualisations have been directed towards managers and researchers [43, 49], rather than developers. This appears not to be an intentional direction and merely a by-product of the nature of metrics, which provide high level aggregate information, rather than fine detail, and so fall more into the domain of managers rather than developers. *In situ* visualisation is a reversal of this trend. In tools such as CoderChrome we specifically direct metrics data towards developers, although managers and researchers may also gain benefits. This because CoderChrome ties metrics data directly to source code. This provides a dramatic broadening in scope of the applicability of software metrics. Some of the specific consequences of this change are itemized below.

- Established product metrics approaches are largely based on static snapshots of source code. This makes them suitable for analysis at various points of development, such as when it is released. *In situ* visualisation, however, is innately dynamic; it updates as source code is edited. This has the potential to transform the way metrics are used, making them a routine consideration for developers as they write code.

- Developers do not have the time or inclination required to use existing visualisation tools [49]. The need to calculate and view metrics outside the normal IDE interface compounds the complexity of the development process. CoderChrome places the visualisation directly within the source code editor, makes metric calculation a continuous process, and provides easy control over the visualisations.

- Existing visualisations provide data that is potentially more useful to those with birds-eye-view of a project. Some examples of this include the CodeCity and tree map based metaphors [34,51]. CoderChrome can provide visualisations of data that are more relevant to a developer.

- Established process metrics tend to be disjoint from product metrics. Measures such as hours worked, number of bugs fixed, and amount of collaboration have traditionally been easier to gather without reference to code structure. CoderChrome allows metrics such as these to be visualised directly on top of code structure, effectively creating a new class of metric that ties together process and product dimensions.

6.2 A Day In The Life

Developers spend their time in a wide variety of tasks (see appendix C), many of which focus on the use of a source code editor. Such tasks include writing new features, debugging, refactoring and bug finding. In this section we provide a series of scenarios in which CoderChrome would be beneficial to a developer.

- A bug report is received detailing a bug in the latest update of the GUI. As a developer on the GUI, you know that feature was working in the previous release. Using CoderChrome, you turn on a Code Age metric showing you which lines of code we edited most recently. This allows to to scan through the code and easily identify the offending changes.
- As a developer, you have found a large and complex method that needs refactoring. In order to assist you, you turn on an Edit-Thrash metric to determine the number of times each line of code in the method has been edited. Any expression or code block that has had many edits, and is therefore been a problem in the past, should warrant more caution when deciding how to refactor.
- As a developer you are adding a new section of functionality. A Code Coverage metric you are running shows that you have no tests that run a particular line of code. You rethink your strategy and develop a test before continuing to add new code.
- As a project manager, interested in the benefits of pair programming and software quality, you turn on augmentations of two metrics - one to show which sections of code were pair programmed and the other to show the number of bugs found in any given section of code. This allows you to easily evaluate which direction is most appropriate for your development team.
- Providing a metric of the code's author and the another of the number of bugs found could provide extremely valuable results for identifying the development style of each developer. This could lead to successful programmers being paired with those who often make mistakes.

7. Future Directions

An important characteristic of this work is that it provide a basis for continued development, this section describes our plans for future work.

7.1 Future Tool Development

In section 4, we discuss the development of our tool, CoderChrome. CoderChrome has already reached a state of development where it is able to be deployed in industrial settings, however, before this can happen metrics calculation tools must be developed. These tools should be developed so they are consistent with the framework established by CoderChrome and in particular provide the ability to update metrics dynamically. This means that they should use the Observer pattern to make them responsive to changes in the editor and update metrics accordingly; CoderChrome in turn will use the Observer pattern to detect the changes and update the display.

There are a range of improvements that could add functionality to the tool. We have already discussed the potential extensibility of the tool in section 4.5. In that section we discussed how the tool could be extended to provide support for additional editor types, IDEs and diagramming types. These extensions have the potential to be very useful and are a major focus of future work. We believe that the two most valuable additions from this extensibility would be the support of Microsoft's Visual Studio IDE, particularly with regards to the C++ and C# languages and an extension to provide overlay support for a round trip UML diagramming tool in Eclipse.

However, there are many other improvements we would like to make to the system. The most basic of these would be the addition of further augmentations to the tool. This would grant the user additional flexibility in making choices about to how to represent metrics data. This would be useful as certain metrics can be more clearly portrayed when using certain types of augmentation. These additions could include further abilities to provide typographical changes (see figure 3.4(g)), better hover techniques (see figure 3.4(j)) and vertical bars (see figure 3.4(l)).

Parallel projects are underway to support metrics in Eclipse, for example the EclipseMetrics plugin [50]. We would like to be able to integrate other metrics sources with our CoderChrome tool, however, this would require changing these projects from static snapshot metrics calculation approach to a dynamic approach.

Our preliminary evaluation revealed the need for improved documentation. While existing documentation is extensive for users of the system, it is fairly limited for those seeking to extend the plugin, especially for those intending to provide new metric sources.

7.2 Future Evaluation

In section 5 we presented our results and current progress in evaluating CoderChrome. Given the promising results that we have so far received, we believe that further evaluation is essential, once suitable metrics calculation tools are available. This section presents an evaluation plan that

looks at the important questions that we seek to answer and the types of evaluation that would be suitable to provide valid answers.

- We anticipate CoderChrome undergoing a series of iterations, these should be mirrored by a series of evaluations to track changes in usability. In addition to ensuring that the tool is robust and performs well, such evaluations would allow us to provide direction for further developments.

- Another concern is the HCI issues involved with this visualisation. These include determining which types of augmentations can be shown together effectively and which may interfere with each other. In addition, we are interested in the number of augmentations we can display simultaneously without impeding the developer, and which types of augmentations work best for this. Finally, we are interested in the correspondences between metrics and augmentations; which augmentations provide the most effective method of displaying a particular metric. To provide answers to these questions, formal quantative evaluations are the most suitable.

- An important evaluation the needs to be completed is to compare the usefulness of *in situ* visualisation, in a tool such as CoderChrome, with other visualisation techniques. This would allow us to determine which types of tasks benefit from *in situ* visualisation. As these techniques are designed to be used in an industrial setting, it makes sense that this evaluation would take that into account. In this case, the most appropriate type of evaluation would be a extended case study involving a number of experienced developers.

- If the visualisations above provide positive results, we will have established a stable and well evaluated framework for the representation of metrics data to developers. In this situation, we can use the tool to perform further evaluations as to the actual benefits of individual software metrics and receive sound results. This is a difficult goal that has never been achieved and would provide a significant advancement in our understanding of software engineering.

8. Conclusion

This project has investigated a new visualisation technique in which software metrics are displayed to users *in situ*; that is to say directly inside a source code editor. With this technique and the tool we have developed, we can display a wide variety of metrics information with a variety of augmentations.

In this project, we have made the following contributions:

- The identification and formalising of a new method of visualising data for software developers.
  - The classification of the types of data, particularly metrics and heuristics that this visualisation could provide.
  - The classification of the types of augmentation that can be used to visualise provided data sets.
- A tool, CoderChrome, that implements this visualisation in a frequently used IDE.
  - The development and implementation of an extensible OO model that describes the relationship between a metric and an augmentation.
  - The identification of the different methodologies for evaluating the visualisation.
  - The development of a system designed for large scale products in real world software engineering environments.
- An initial informal evaluation of the tool.
- The discussion that highlights the enormous benefits that this tool can provide software developers.
- A platform and future directions for continuing research in this area.

This technique has the potential to provide dramatic benefits to software developers. By presenting a visualisation within the view that these developers are familiar with, we are providing a system that could allow developers to use useful metrics data to solve the complex problems associated with large scale software development. We are confident that this tool will increase the productivity and satisfaction of the developers who use it.

Bibliography

[1] Anonymous. *About The Eclipse Foundation*. The Eclipse Foundation, http://www.eclipse.org/org/, 2009.

[2] Anonymous. *Clover: Code coverage analysis*. Atlassian, http://www.atlassian.com/software/clover/, 2009.

[3] Anonymous. *Code Collaborator*. SmartBear Software, http://smartbear.com/codecollab.php, 2009.

[4] Anonymous. *Eclipse Marketing Resources - Research Papers*. The Eclipse Foundation, http://wiki.eclipse.org/Marketing_Resources#Research_Papers_of_Interest, 2009.

[5] Anonymous. *Eclipse Wiki*. The Eclipse Foundation, http://wiki.eclipse.org/Main_Page, 2009.

[6] D. Astels. *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference, 2003.

[7] A. Aula and V. Surakka. Auditory emotional feedback facilitates human-computer interaction. *People and computers XVI: memorable yet invisible*, page 337, 2002.

[8] V.R. Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th international conference on Software engineering*, pages 442–449. IEEE Computer Society Washington, DC, USA, 1996.

[9] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] B.W. Boehm. *Software engineering economics*. Prentice-Hall Englewood Cliffs (NJ), 1981.

[11] B.W. Boehm. Get ready for agile methods, with care. *Computer*, pages 64–69, 2002.

[12] F.P. Brooks. The Mythical Man-Month: Essays on Software Engineering. Adison Wesley, 1995.

[13] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[14] N. Churcher and W. Irwin. Informing the design of pipeline-based software visualisations. In *APVis '05: proceedings of the 2005 Asia-Pacific symposium on Information visualisation*, pages 59–68, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[15] M.F. Cowlishaw. LEXX - A Programmable Structured Editor. *IBM Journal of Research and Development*, 31(1):73–80, 1987.

[16] N.E. Fenton and M. Neil. Software metrics: roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 357–370, New York, NY, USA, 2000. ACM.

[17] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[18] A. Ford and A. Roberts. Colour space conversions.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[20] M. Harward, W. Irwin, and N. Churcher. In situ software visualisation. *Submitted to Australian Software Engineering Conference*, 2009.

[21] B. Henderson-Sellers. Object-oriented metrics. In *Proceedings of the eleventh international conference on Technology of object-oriented languages and systems*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993.

[22] D. Hendrix, J.H. Cross, and S. Maghsoodloo. The effectiveness of control structure diagrams in source code comprehension activities. *Software Engineering, IEEE Transactions on*, 28(5):463–477, May 2002.

[23] J.D. Herbsleb. Global software engineering: The future of socio-technical coordination. In *FOSE '07: 2007 Future of Software Engineering*, pages 188–198, Washington, DC, USA, 2007. IEEE Computer Society.

[24] J. Highsmith and A. Cockburn. Agile software development: The business of innovation. *Computer*, pages 120–122, 2001.

[25] W. Horton. Top ten blunders by visual designers. *SIGGRAPH Comput. Graph.*, 29(4):20–24, 1995.

[26] W. Irwin and N. Churcher. Xml in the visualisation pipeline. In *VIP '01: Proceedings of the Pan-Sydney area workshop on Visual information processing*, pages 59–67, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[27] W. Irwin and N. Churcher. Object oriented metrics: precision tools and configurable visualisations. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 112–123, 2003.

[28] G.H. Joblove and D. Greenberg. Color spaces for computer graphics. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 20–25, New York, NY, USA, 1978. ACM.

[29] C. Jones. *Estimating software costs: bringing realism to estimating*. McGraw-Hill Osborne Media, 2007.

[30] C. Jones. *Applied software measurement*. McGraw-Hill Osborne Media, 2008.

[31] P. Kahn and K. Lenk. Design: principles of typography for user interface design. *interactions*, 5(6):15, 1998.

[32] M. Keil. Pulling the plug: software project management and the problem of project escalation. *Mis Quarterly*, pages 421–447, 1995.

[33] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 131–142, New York, NY, USA, 2008. ACM.

[34] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The visual code navigator: An interactive toolset for source code investigation. In *Proc. IEEE InfoVis*, volume 5, pages 24–31. Citeseer, 2005.

[35] G.W. Meyer and D.P. Greenberg. Perceptual color spaces for computer graphics. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, page 261. ACM, 1980.

[36] G.C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE software*, pages 76–83, 2006.

[37] B. Neate, W. Irwin, and N. Churcher. Coderank: A new family of software metrics. In *Software Engineering Conference, 2006. Australian*, page 10, 2006.

[38] D.L. Parnas. Software engineering or methods for the multi-person construction of multi-version programs. *Lecture Notes in Computer Science. Programming Methodology*, 1974.

[39] L. Powers and M. Snell. *Microsoft Visual Studio 2008 Unleashed*. SAMS Carmel, IN, USA, 2008.

[40] A.J. Riel. *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.

[41] V. Roto and A. Oulasvirta. Need for non-visual feedback with long response times in mobile hci. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 775–781, New York, NY, USA, 2005. ACM.

[42] D. Rubinstein. Standish group report: Theres less development chaos today. *Software Development Times*, 1, 2007.

[43] J. Segal, A. Grinyer, and H. Sharp. The type of evidence produced by empirical software engineers. In *REBSE '05: Proceedings of the 2005 workshop on Realising evidence-based software engineering*, pages 1–4, New York, NY, USA, 2005. ACM.

[44] A. Sharma. *Understanding color management*. Cengage Learning, 2003.

[45] A. Sharp. *Smalltalk by example*. McGraw-Hill, 1997.

[46] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java developer's guide to Eclipse*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[47] M. Stokes, M. Anderson, S. Chandrasekar, and R. Motta. A standard default color space for the Internet–sRGB. *Microsoft and Hewlett-Packard Joint Report, Version*, 1, 1996.

[48] P.H. Sydenham, N.H. Hancock, and R. Thorn. *Introduction to measurement science and engineering*. John Wiley & Sons, 1989.

[49] M. Umarji and C. Seaman. Why do programmers avoid metrics? In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 129–138, New York, NY, USA, 2008. ACM.

[50] L. Walton. *EclipseMetrics plugin*. State Of Flow, http://eclipse-metrics.sourceforge.net/, 2009.

[51] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 921–922, New York, NY, USA, 2008. ACM.

[52] C. Wohlin, M. Höst, P. Runeson, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Pub, 2000.

[53] Y. Yanagida. Olfactory interfaces. *HCI Beyond the GUI: Design for Haptic, Speech, Olfactory, and Other Nontraditional Interfaces*, pages 267–290, 2008.

Appendices

A. View Control Panel

The view shown in figure 4.2 provides the following functionality:

- A textual description of the current state of the tool. This includes a representation of the last time updated to allow the user to comprehend the validity static data sets.
- The ability to access the preference pages and CoderChrome's help documentation.
- CoderChrome supports three main levels of activity for mappings - undeclared, declared and active. Only declared and active mappings are shown within the view. This has been designed to provide specifically for cases where a large number of mappings are present. The view provides the ability to easily toggle a mapping between a declared and active state. It also provides a set of controls for controlling those mappings visible (declared) within the view.
- Each visible mapping shows a textual representation of the name of the mapping and the associated names of the the metric and augmentation that they relate to.
- The view control also provides a global toggle for toggling between active and declared for all mappings in the view.

This view is built directly into the Eclipse framework and, once the tool is installed, can be accessed by clicking "Window", then "Show View", and then selecting the Metrics Overlay view. By default, this will then be visible in the lower pane of the current workspace.

B. Preference Pages

Below are example of the four different preference pages.

B.1 General Preferences

Figure B.1 is a screen shot of the general preference page. It provides access to the the other preference pages and provides system wide configuration settings.

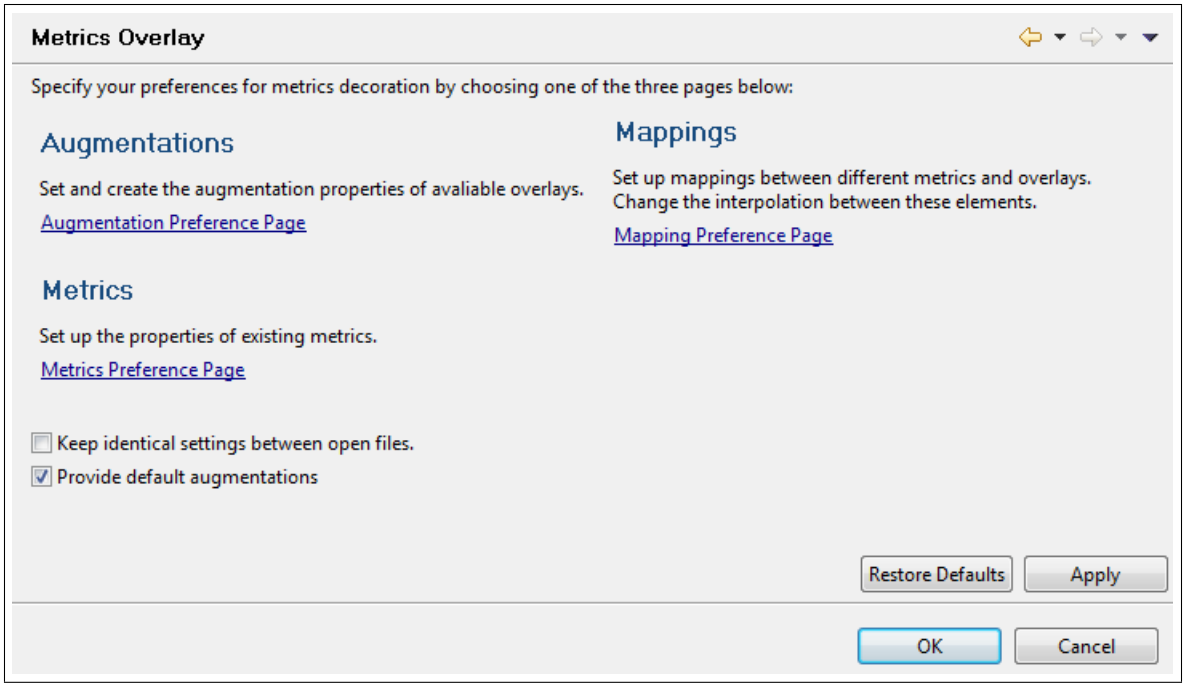


Figure B.1: The general preference page

B.2 Metrics Preferences

An example of the metrics preference page can be seen in figure B.2. Common among all of the preference pages, are the ability to select the current item, see and change its name, and provide the functionality to update or add a new metric.

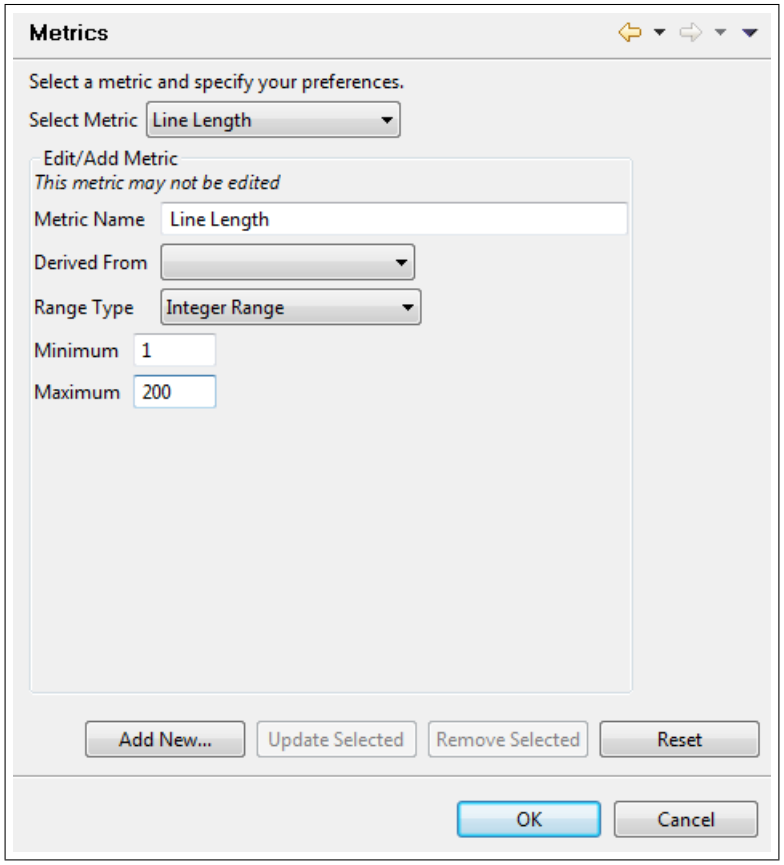


Figure B.2: An instance of a metrics preference page

B.3 Augmentation Preferences

In figure B.3, the augmentation preference page is visible. This page allows users to create new or alter existing augmentations, changing all of the settings specific to each augmentation.

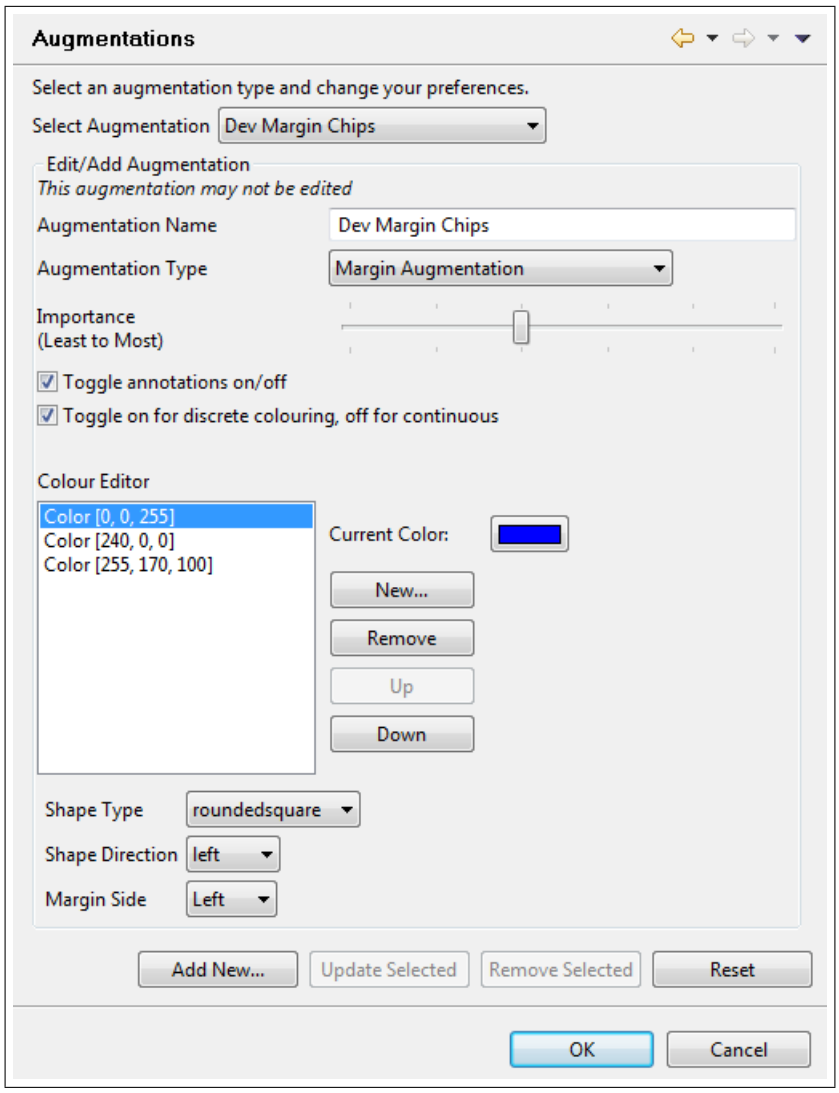


Figure B.3: An instance of a augmentation preference page

B.4 Mapping Preferences

The mapping preference page, seen in figure B.4, provides the ability to create new combinations of existing augmentations and metrics. Existing mappings can be updated or new mappings created.

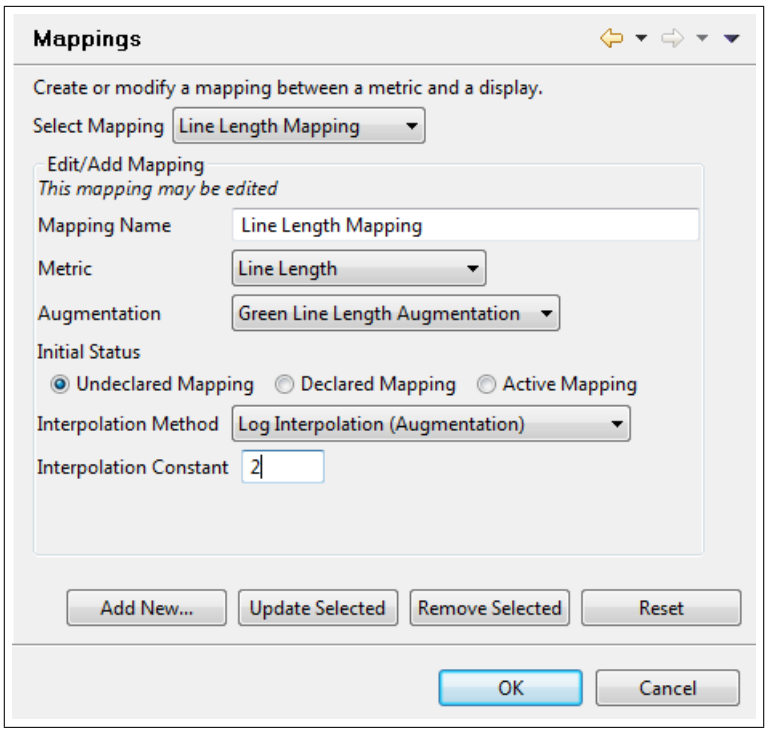


Figure B.4: An instance of a mapping preference page

C. Developer Activities

In this section we detail a list of potential developer activities. Some developers may spend greatly varying quantities of time on each of the following tasks:

- Communication and Meetings
  - With other team members
  - With managers
  - With clients and customers
  - With less experienced developers
- Architecture and Design
  - Understand requirements
  - Gathering/researching requirements
  - Designing system components/classes/algorithms
- Development
  - Setting up a development environment and code repository
  - Coding new functionality
  - Refactoring
  - Debugging
- Testing
  - Unit testing
  - Integration testing
  - Code review
  - User interaction testing[1]
- Deployment
  - Producing new releases
  - User documentation
- Documentation

[1] For example, the testing of GUIs, using heuristic testing, and collecting and evaluating user data to optimize a system.

  - Producing system documentation[2]
  - Documenting and commenting code
- Maintainence
  - Fixing bugs
  - Adding new functionality
- Other
  - Further business operations[3]

[2]This has become a less practiced task with the development of agile methods.

[3]For example, accounting, marketing, and recruitment. These tend to occur more frequently in smaller operations where they lack dedicated staff.

D. Previous Paper

The following paper [20] has been submitted to the 2010 Australian Software Engineering Conference (ASEC).

In Situ Software Visualisation

Matthew Harward, Warwick Irwin, Neville Churcher
Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
neville.churcher@canterbury.ac.nz

***Abstract*—Software engineers need to design, implement, comprehend and maintain large and complex software systems. Awareness of information about the properties and state of individual artifacts, and the process being enacted to produce them, can make these activities less error-prone and more efficient. In this paper we advocate the use of code colouring to augment development environments with rich information overlays. These *in situ* visualisations are delivered within the existing IDE interface and deliver valuable information with minimal overhead. We present CODERCHROME, a code colouring plug-in for Eclipse, and describe how it can be used to support and enhance software engineering activities.**

***Index Terms*—software visualisation; software metrics; code colouring;**

I. INTRODUCTION

Software engineering, after four decades, is still a discipline rich in challenging problems. Its core business, the multi-person construction of multi-version programs [1], involves software which is both large and complex.

Measurement is central to all forms of engineering and software engineering is no exception. It is important to be able to measure the artifacts produced as well as the processes used to produce them, in order to control and improve quality and efficiency. However, the size and complexity of the resulting data sets means that it is not straightforward to make effective use of such measurements even where they are available.

Software process models have been developed to describe software development activities in terms of tasks, decisions intermediate products and other quantities. When a process is enacted data can be recorded about details such as tasks, scheduling, resource allocation and time spent.

Software metrics allow data to be recorded about the process and products [2]–[5]. Process metrics help software engineers understand where time and resources have been expended, estimate project planning quantities such as time to completion and to predict defect densities.

Product metrics include quantities such as size and complexity of components. While product metrics have been around for a long time [6]–[9] there remain a number of outstanding obstacles to their widespread adoption and effective use.

Precise definitions of individual metrics and tools for accurate and complete data collection are required. Our previous work has included an approach which allows metrics to be defined in terms of the standard grammars of the programming languages employed; it supports the development of powerful tools for the acquisition, processing and exchange of metric data [10], [11].

There is an 'impedance mismatch' between the low-level metrics data which is recorded directly and the more abstract quantities which are of interest to software engineers. The latter must be modelled by proxies which are indirectly computed from the fundamental measurements. For example *Comment density* ($\frac{CLOC}{NCLOC}$) *is a measure of* Self-descriptiveness *which is one aspect of* Reusability*, which is one element of* Product transition *which is one component of* Quality. Established quality models [12] are often insufficiently detailed to be useful in practice.

No single metric is sufficient to describe software adequately and in practice a number of metrics is required in order to cover the significant dimensions of the software products and process enaction. Interpretation—finding the information in the data—is a significant challenge. Although the use of metrics allows us to take an abstracted 'bird's eye' view of software, even these greatly simplified views are extremely complex for large systems. We still desperately need ways of supporting software engineers as they struggle to comprehend all the relevant factors in development activities.

In reality, software engineering is rendered in shades of grey rather than black and white: solutions are often the result of balancing competing forces. Examples include the GoF design patterns [13] and Riel's heuristics [14]. Consequently, we need to support the visualisation of 'softer' heuristics as well as more conventional metrics and have made some steps in this direction [15].

Additional complexities arise where principles or heuristics conflict and developers must be supported as they explore the solution spaces and make optimal choices. Our desire to support developers in such activities is one of the main motivations for the work reported in this paper.

In our previous work we have addressed these issues in a number of ways. Rather than adhering rigidly to a specific set of metrics, we advocate flexible configuration at the detailed level (e.g. suppressing the display of private methods) in order to facilitate exploration of metric data sets according to the task at hand. Software visualisations such as class clusters [10] provide a more 'holistic' impression of a system. Metrics are employed because the effects of the scale and complexity of the software under consideration: there is an inevitable loss of information since not everything can be measured and a subjective element in the selection of the metric set

used. In this respect they are like the star ratings used for movie reviews in that, although they suffer from various flaws, they are nevertheless of practical value. We argue that visualisation moves us closer to having the best of both worlds. Visual metaphors, rather than numeric proxy measures, allow entire systems to be presented, providing valuable context information. We can then deliver a blend of intrinsic system structure for context, through computed geometry, and relevant detail by decorating the geometry to reflect metrics.

Modern development practices add further complexity. For example, revision control systems allow detailed data to be gathered about evolving systems: examples are monitoring the changes in complexity of methods, the resources expended in refactoring and the contributions of individual developers. Agile process enactions generate fewer formal artifacts than more traditional processes but monitoring and steering them are equally important.

It would be typical to collect some tens of metrics. The software code base may be very large—it might consist of millions of lines of source code, thousands of classes and tens of thousands of methods. Measurements will be taken successively—at intervals or at version control system check in time. Consequently software engineers face information overload problems when trying to interpret or explore such data sets.

Software development covers a range of activities including design, testing comprehension, refactoring, debugging and coding. Each has its own information needs, and it is desirable to support as much as possible from a consistent set of environment features. Metrics provide information of relevance in these and other activities. We aim to provide metrics-based information to developers *while they are carrying out these tasks*. We are not primarily concerned here with the use of metrics in project management or post-mortem project analysis.

While techniques such as those discussed above can help considerably, there are limits to the number of information sources that can be provided in the development environment and their effectiveness as this number increases. We therefore seek ways to use more effectively for visualisation the essential components which will already be present in the interface.

We believe that individual developers can be better supported by the provision of richer information about the current working set or artifacts but also about the wider context in which they are working. To achieve this without introducing distracting elements we advocate using the essential interface elements, such as text editors, which are already the focus of the developer's attention. We use the term *in situ* visualisation to describe this approach.

In this paper, we contend that the provision of *in situ* visualisations such as code colouring can benefit developers by providing richer information resources without occupying valuable display space. We achieve this by providing visualisations where the developer's attention is already focused.

The remainder of the paper is structured as follows. In the next section we discuss *in situ* visualisation and indicate how it can assist developers. Our approach is described in Section III and Section IV contains a description of the CODERCHROME Eclipse plug-in we have implemented, followed by some reflections in Section V. Finally, we present our conclusions and outline our ongoing work in Section VI

II. IN SITU VISUALISATION

Awareness and exploration are key factors in enabling software engineers to make effective use of information derived from software metrics. The activities involved in comprehension of software require management of many elements and their inter-relationships. Some information items are essentially local (to a class, design pattern, package, ...) while others are intrinsically distributed (scope/visibility, method invocation, ...) so it is necessary to have both detailed information about specific elements but also to maintain awareness of their connections to the remainder of the system. This 'focus plus context' problem is well known in information visualisation.

Software engineers often require more than just 'read only' information in order to perform development tasks. It is valuable to have facilities supporting exploration so that developers can drill down into more detailed information, follow trails suggested by previous information or obtain different perspectives.

Modern software engineering occurs in environments featuring complex tools with large feature sets and complex human-computer interfaces. The architectures of such tools generally allow extensibility via plug-ins which can be selected according to the specific supplementary or alternative functionality required. This allows flexible customisation of the working environment but the number, range and consistency of interface elements is likely to increase.

Eclipse (http://www.eclipse.org) is an example of a typical modern IDE. Figure 1 shows a typical display state presented to developers. Significant amounts of the available screen real estate are taken up with menubars, tool bars, navigation panes, tabbed panes for transcript or additional data, help panes and so on. Consequently the space available for the core business of editing code is at a premium. Furthermore, this must usually be shared among a number of open files in a working set.

It is clear that the source code being edited often occupies only a small portion of the display. While it is possible to dynamically reconfigure the display to hide or relocate some components and maximize others, this introduces further difficulties as the user must deal with rapid context switches.

Some IDEs feature UML or other diagrams and in some cases [16] these are active, supporting true round trip engineering where changes may be made via text or diagram editor. This introduces yet more elements into the interface.

Team development is supported in a number of ways. Most IDEs support interaction with one or more revision control systems. Individual developers check out files from a repository, make their changes and attempt to check them back in. If another developer has checked out and modified another

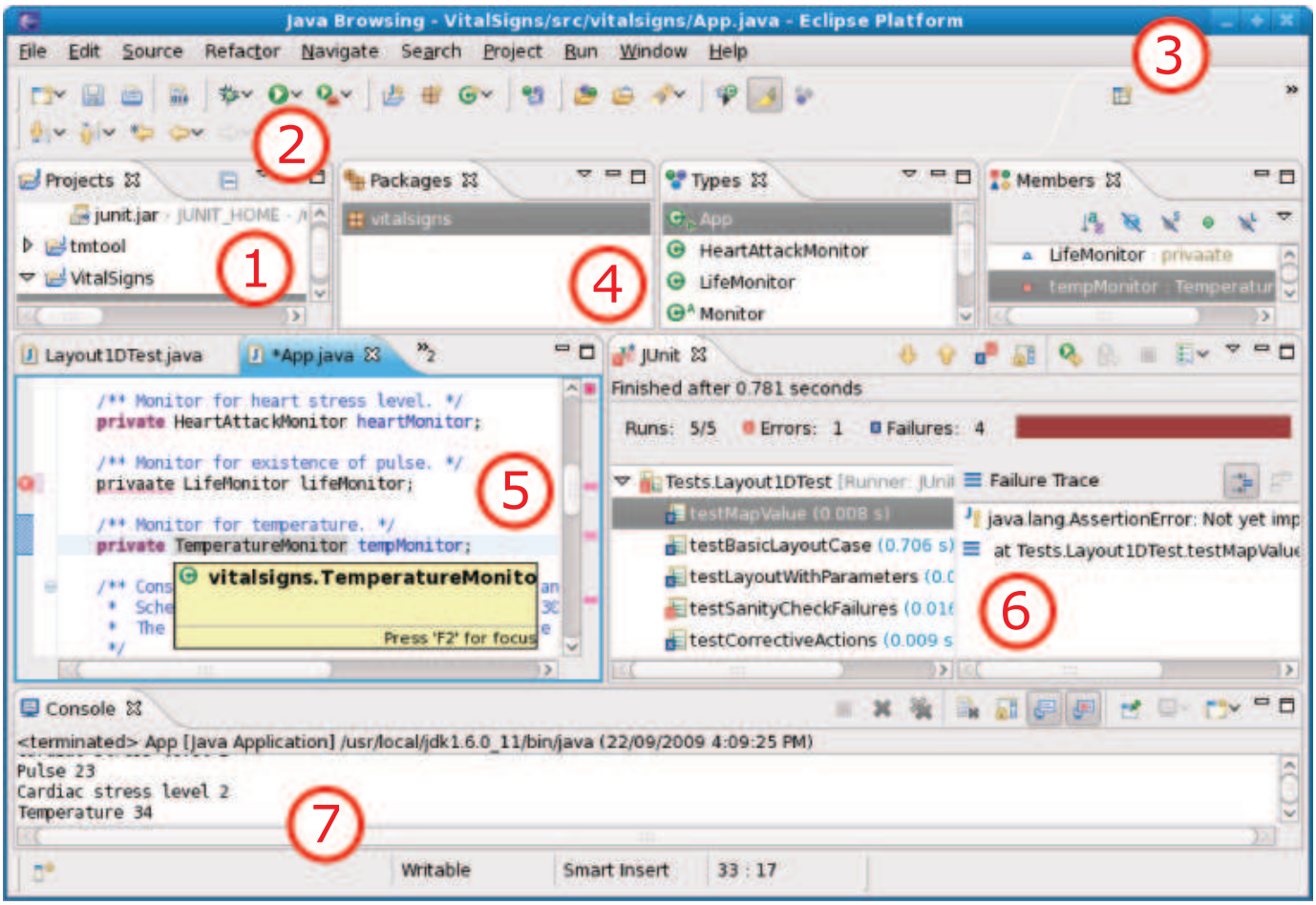


Fig. 1. The Eclipse Java Editor. 1) A project explorer for viewing and accessing the structure of the Java application. 2) Tool bar containing commands for directly running the written application. 3) Selection of the current perspective. 4) An explorer for examining the packages, types and members in the current project. 5) The editor, with syntax highlighting and code completion. 6) Integrated unit testing facilities. 7) Other views are available, for example console output.

copy of the file then the potentially conflicting changes must be integrated before the file can be committed.

A further level of sophistication occurs in real-time collaborative groupware for software engineering [17]–[23]. Such systems allow several developers to make concurrent changes to files. Users are provided with sufficient awareness of the locations and activities of others to allow conflicts to be addressed as they arise.

Essential activities such as those described above can be better supported where the development environment includes mechanisms for providing awareness and supporting exploration. A significant challenge is how best to provide these in an already cluttered interface.

In this paper, we contend that the provision of *in situ* visualisations such as code colouring can benefit developers by providing richer information resources without occupying valuable display space.

The *in situ* visualisations we propose have a clear relationship with ambient information and software visualisation techniques [24]–[28]. These include the use of sound, desktop background image and window decorations to provide information in an unobtrusive manner which does not detract from foreground tasks.

In our present work we are concentrating on the same thing as the developer is—the source code in the text editor. However, we anticipate extending our work into the wider ambient visualisation domain. Indeed, we have already considered (and discarded!) olfactory interfaces [29] to provide literal code smells.

III. Code colouring augmentations

The idea of code colouring is not new. Syntax highlighting has been a feature of source code editors for some time. This typically involves the use of foreground colour and font to indicate programming language keywords, comments and literal strings. Debuggers often use reverse video or background colour to denote the location of the current breakpoint.

Small symbols may be placed in the margins or scroll bar areas to denote such things as the location of breakpoints, errors or possible areas requiring refactoring. Examples are visible in Figure 1.

Seesoft [30]–[32] is arguably the best known use of code colouring in software visualisation. The display essentially shows a view from far away of file listings pinned to a wall alongside each other. Individual lines are coloured according to the value of some quantity but are too small to read, although significant features may often be recognised by characteristic patterns of line lengths. Applications include fault localisation [33] and steering visualisation design [34].

Questions a developer might have while working on a piece of code include:

- Which team member last changed it?
- How long is it since it was last refactored?
- How many defects have been identified in it?
- How many times has it been edited since the last release?
- How urgently does it need refactoring?
- What is the distribution of cyclomatic complexity?

Fig. 2. Code colouring in SeeSoftLike [34]

In earlier work [34] we developed *SeeSoftLike*, a standalone tool based on the Seesoft concept, in order to help address such questions.

Figure 2 shows two views of the same Java source code file: the lines of each are coloured identically. The window at the left of the figure shows the entire file as seen from afar. The window at the right of the figure shows a small part of the file at normal size, providing a simple *in situ* visualisation. Our current work is motivated by the desire to extend this concept to realistic development environments such as eclipse.

In previous work [18] we described a simple code age editor which showed 'fresh' code in darker shades and older code 'faded' away to lighter shades. This tool was part of a real-time software development environment and allowed individual developers to see what changes were being made by others as well as by themselves.

In the current work we generalise this approach. One key difference is that in our approach, code remains at its natural font size and so is readable and may be updated. Our approach allows more that one additional quantity to be displayed and no separate interface elements are required.

The idea, as indicated in Figure 3a, is to allow any available quantity, which may be a single metric or an arbitrary function of several metrics, to be selected as the basis for colouring. Some data will be available from a repository of historical data and may be produced by external tools. Other data will arise dynamically, from the Eclipse framework itself or as a result of user actions. Figure 3b shows the process in more detail and Section IV includes discussion of implementation issues.

A mapping stage includes selecting the particular *in situ* visualisation required. It also specifies the transformation from the selected metric(s) (which will include quantities with ration, interval, ordinal or category scales) to an appropriate colour range. Simple linear mapping functions may suffice in some cases, but non-linear functions (such as tanh or log) are needed where the selected metric may have potentially infinite values.

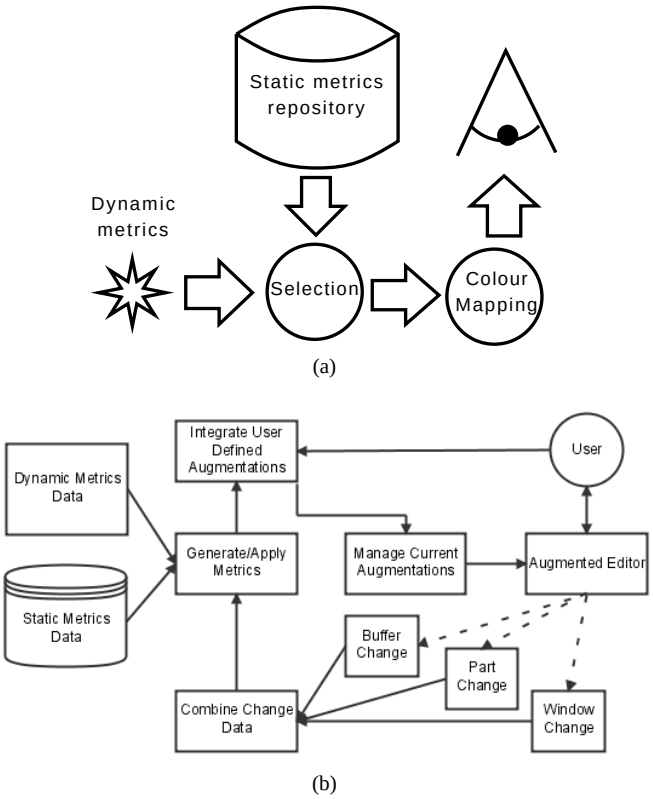


Fig. 3. Schematic views of CODERCHROME architecture

We distinguish two modes for deploying *in situ* visualisations. In static cases we are working with a snapshot of the

code and need not consider continually updating the display. In dynamic situations it is necessary to update the display(s) in response to events. These may be internal, such as edits to the file, or external, such as receiving data from an external tool.

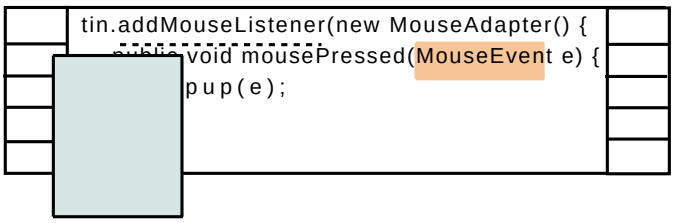


Fig. 4. Parts of the line available for augmentation with *in situ* visualisation

Figure 4 shows the elements of our augmentation model. The line text, including any any pre-existing foreground colouring or font selection, is unchanged. This allows CODERCHROME to co-exist with typical syntax highlighting as used in IDEs such as eclipse. In such situations the colour scheme selected should avoid colours close to those used for foreground text highlighting in order to preserve readability.

The background colours of each line, together other augmentation elements, are managed by CODERCHROME according to the current mappings.

Our model includes the following augmentation types, most of which have already been implemented in our prototype. Examples and further details appear in Section IV.

- The background colours of each line are managed by CODERCHROME according to the current mappings. For example, the lines may be mapped to a colour scheme which shows more recent code with a brighter background, and older code with a darker background, and which provides good contrast with the foreground colour scheme (set using eclipse's own preferences).
- An augmentation in the form of a glyph, which may be a simple colour chip or a more active component, may be placed at the beginning of each line to allow a further quantity to be displayed.
- Similarly, a further augmentation may appear at the end of each line. These might be mapped to quantities of interest such as the original author, time since last test, complexity or priority for refactoring.
- Colouring may be applied to individual regions, which may be ranges of characters within individual lines or may extend across multiple lines. This might be used to indicate the presence of code smells [35].
- The use of underlining to indicate possible spelling, grammar or style violations is familiar across a wide range of applications. We generalise this to allow similar soft feedback about design principles and heuristics. For example, a red underline might appear when a cyclic dependency is created. If the underlying data is available, this approach allows relatively unobtrusive code critics and subliminal alerts to be supported.
- Hovering, or right-clicking, over an augmentation leads to further detail. This might be in the form of a tooltip, but could provide more complex information (e.g. in the form of a bar chart popup).
- Augmentations could also provide links to other artifacts (bug reports, test cases, notes for refactoring, ...) but we do not view these as central to our *in situ* approach.

Our work is intended to lead to improvements in the tool sets available to developers and it is useful to view these alongside generally accepted aspirations reported by others. In one recent evaluation of software visualisation tools applicable to corrective maintenance [36] a number of desirable features were identified. Although the specific context of that work was a little narrower than ours, the categories used are more widely applicable. We now use some of them to discuss briefly the merits of our code colouring approach.

Scalability: The *in situ* nature of code colouring enables high scalability since no additional display space is required.

Integration: Provision of a well-defined interface to CODERCHROME enables it to use data from a wide range of sources. Our aim is to be able to perform colouring based on data from the eclipse environment, other plug-ins and external tools.

Query support: We do not provide a full-featured *ad hoc* query interface, but do provide a configuration interface which allows queries to be defined and refined to support exploration. While complex queries such as "Show me the methods written by Steve or Mary last year and which have failed more than 3 tests this month but which have cyclomatic complexity greater than 15 and colour them by the number of hours spent refactoring them" could be supported, we see such queries as better suited to software visualisation techniques other than *in situ* visualisation.

Refactoring: Providing support for design and refactoring tasks is one of our main aims. Our *in situ* augmentations allow various kinds of annotations relevant to such activities (priority for refactoring, code smells, design pattern rôles, ...) to be made available.

Debugging support: Colouring can be used to indicate relevant information such as the density and frequency of breakpoints in code elements. How often breakpoints are set, by whom and how long they remain are quantities of potential relevance to developers.

IV. CODERCHROME IMPLEMENTATION

In this section we describe the tool, CODERCHROME, which we have developed as a prototype to explore the potential of *in situ* visualisations.

It provides an extensible model for directly overlaying source code with augmentations that provide *in situ* representations of metric data. Our long term plans include implementation in a range of IDEs and other tools for various programming languages. However, the initial development reported in this paper has focused on integrating the system into Eclipse as a plug-in.

While many IDEs are available, a small number dominate the industry and education sectors. Eclipse and Visual Studio are notable examples. Our interests include both software engineering education and the support of practitioners and Eclipse allows us to address both from a single platform. Eclipse also has the advantage of providing an open source platform that caters for a wide variety of languages including Java, the focus of much of our previous work.

The system provides augmentations directly into the Eclipse Java source code editor. The approach used can also be used for other languages supported by eclipse. The augmentations appear as colours and symbols in the editor that supplement the existing features provided by Eclipse. Each type of augmentation relates to a quantity derived from metrics data as specified by a particular mapping.

Individual mappings associate one or more specific quantities with the augmentation chosen to deliver the *in situ* visualisation.

Figure 7 shows Eclipse with a number of *in situ* visualisations enabled. Comparison with Figure 1 shows that the overall appearance of the user interface is similar to that of 'normal' eclipse.

Users have access to a visual representation of the available mappings in the form of a view (visible at the bottom of Figure 7) in the Java development perspective of Eclipse. Figure 5 shows a simple example. The corresponding display state is shown in Figure 7.

The view, shown in Figure 5, includes a table which shows the different mappings that are currently available—five in this case. Each individual mapping can be toggled on/off by clicking the corresponding check box and each of the components of the mapping can be editing by opening an edit preferences page by clicking on the corresponding cell. The preference pages provide full control over the visible augmentations, the ranges of the metrics data and how they are related by the mapping.

This includes the functional form of the transformation from data to colour (e.g. for mapping LOC values in the range $[0..\infty]$ to colours in a selected range.

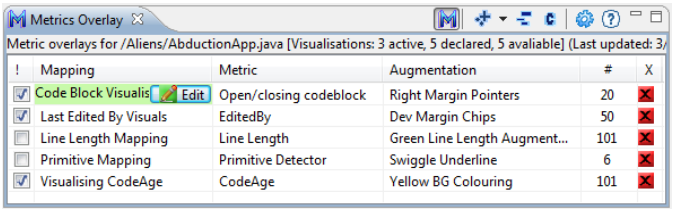


Fig. 5. Metrics overlay selection and CODERCHROME configuration

Figures 8, 9, 10 and 6 show more detailed views of the visualisations specified by these mappings.

The *in situ* visualisation corresponding to the selections of Figure 5 appears in Figures 7. It includes contributions from each of the active mappings.

The background colour of each line in the editor indicates the corresponding value of a code age metric. The corresponding mapping specifies details of the domain of the quantity indicating age and also of the colour scheme to be used. For example, the code age might be expressed in terms of version numbers from a version control system such as subversion, or on a linear scale such as the day last edited. The mapping to a specific colour scheme might include selecting a palette which avoids conflicts with current syntax highlighting settings. It will also specify whether the mapping is linear or whether a non-linear mapping (such as $tanh(\alpha x)$) is required.

```
27    public AbductionApp(String[] names) {
28        randomGenerator = new Random();
29
30        // Make the list of people.
31        people = new ArrayList<Person>(names.length);
32        for (int i = 0; i < names.length; i++)
33            people.add(new Person(names[i]));
34
35        // Make the list of invaders.
36        int numInvaders = randomGenerator.nextInt(10);
37        invaders = new ArrayList<Invader>(numInvaders);
38        for (int i = 0; i < numInvaders; i++)
39        {
40            Invader ship = makeInvader();
41            invaders.add(ship);
42            ship.watch(people);
43        }
```

Fig. 6. Augmentations can be mixed and matched as the user desires. This has the potential to increase a user's exploratory power over their own codebase.

A second augmentation is delivered via the colour chips at the left of each line. The mapping specifies that each colour corresponds to a particular individual—the developer who last edited the corresponding line. This is an example of how nominal and ordinal data can be accommodated alongside interval or ratio scale data.

Figure 9 shows the use of underlining—in this case indicating primitive types—to deliver *in situ* visualisations with minimal difference from the normal Eclipse interface.

Figure 10 shows left-margin colour chips indicating most recent editor. It also shows right-margin augmentations highlighting the block structure, illustrating how existing syntax highlighting can be supplemented (e.g. by using coloured augmentations to show nesting depth).

The current implementation of CODERCHROME provides a variety of augmentation types and we plan to continue adding more options. Currently the available techniques include:

- Background code colouring of each line of code in the document. This is the simplest form of code colouring and is the result of a line based mapping. Figure 8 shows an example.
- Section colouring of each piece of text in the document. This is a character-by-character based mapping. While this allows sections within individual lines to be highlighted, it is more usefully applied in our context to larger units such as blocks or methods.
- Different types of coloured underlines that can be applied to any piece of text in a document. The different underline types include a squiggle, a single or a double line. This supports colourings such as the 'wiggly green underline' familiar from word processing applications.

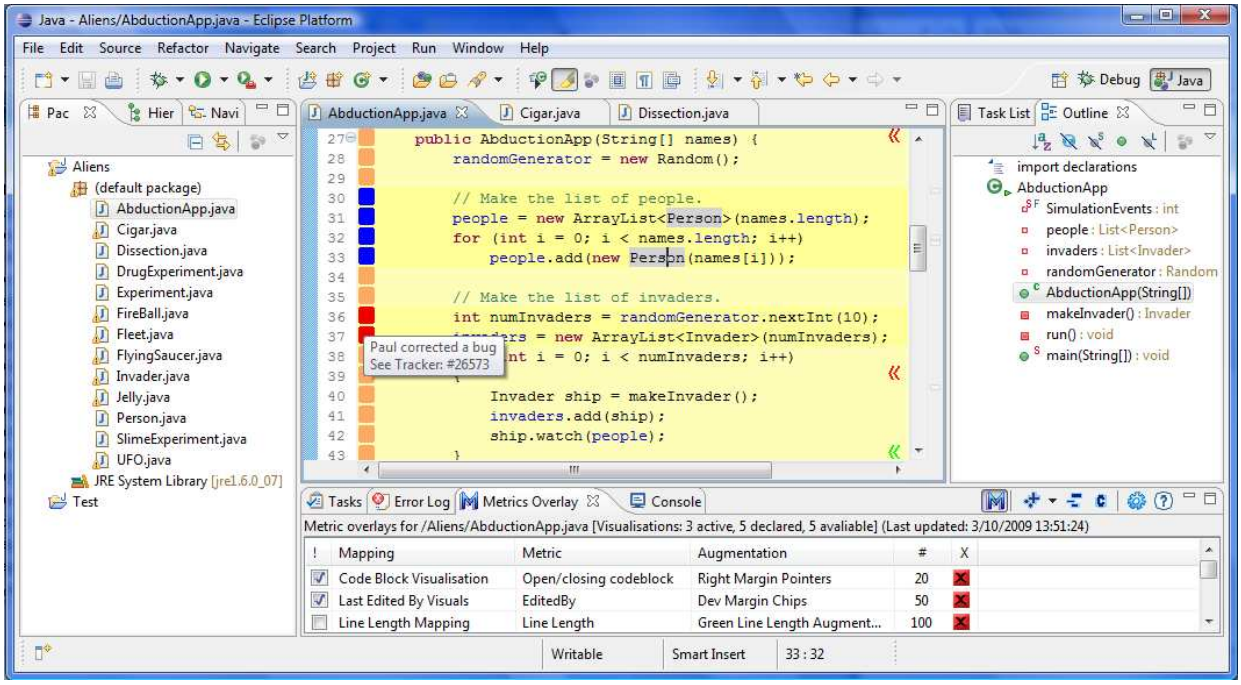


Fig. 7. Eclipse editor with *in situ* visualisations specified in Figure 5

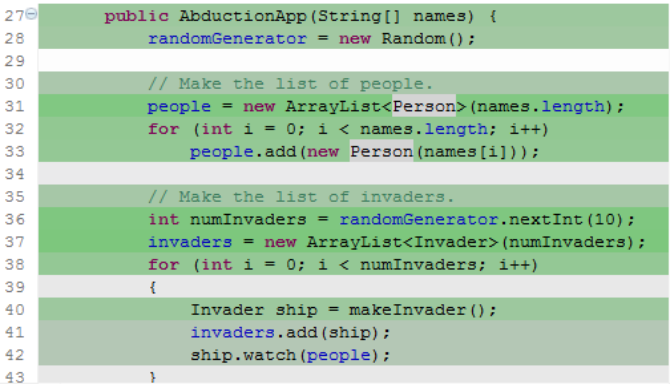


Fig. 8. Different types of background colouring mechanisms can be used. This example shows a line length metric

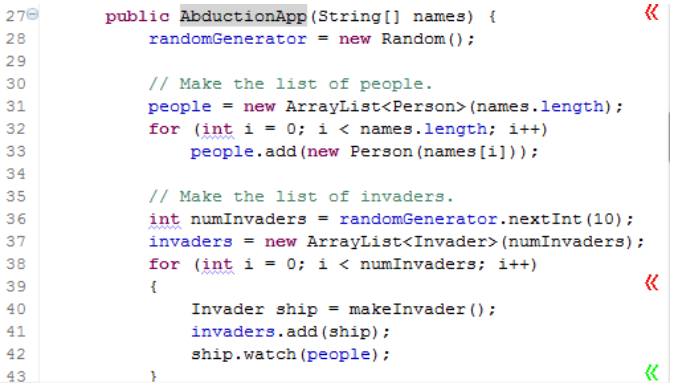


Fig. 9. Augmentations can be very subtle. In this example, primitives are underlined and the start and end of code blocks are indicated by a colour chip in the right margin.

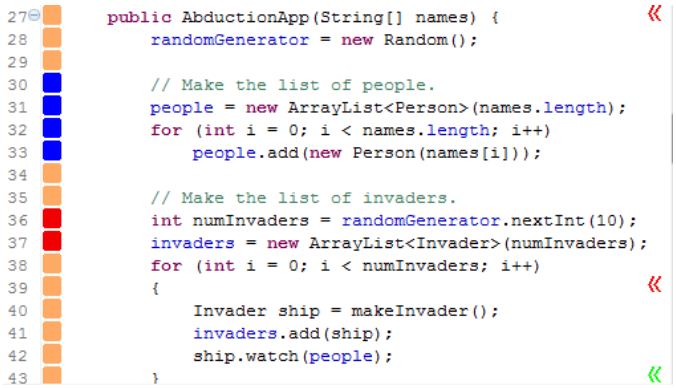


Fig. 10. Colour chips indicating the author and code blocks are displayed here

- Coloured chips of different shapes in the left and right margins. These can also be fitted with tool tip style annotations to provide additional information. A simple example is shown in Figure 7. The (red) colour chip indicating that lines 36–7 were last edited by Paul. Hovering over the chip provides further detail about the change.

While there are many potential candidates to include, such as colour gradients and transparency, it is necessary to resist the temptation to add 'bling' which will dominate the user's attention, diminishing the advantages of *in situ* visualisation.

Our approach supports exploration by presenting combinations of data and allowing the user to draw inferences and investigate patterns. Figure 7 provides an example; in this case that Paul was fixing bugs recently in the file

`AbductionApp.java`.

The system allows any instance of a particular metric to be displayed by providing a mapping between a metric and an augmentation to the source code editor. In order to display data of a wide range of metrics, a generic framework has been designed that provides this functionality. This specification defines a metric in terms of the range it occupies and the current data points of that metric. By allowing ranges to be interchanged, different views of the same data can be achieved.

Eclipse is itself primarily developed in Java and provides a comprehensive development environment for developing applications in Java—the Java Development Tooling (JDT) plug-in.

The Eclipse IDE interface consists of a number of modular components, as can be seen in Figures 1 and 7. Of particular interest in the current work is an editor pane in which source code can be modified. Tabbed panes allow easy navigation between open documents. In addition, a number of views fill the rest of the workbench interface. These views allow the reporting of errors, display task lists, break points, navigation within a project and the properties of an item to be examined and modified. Perspectives allow different combinations of editors and views for different types of development or the use of different languages.

JDT provides plug-ins for Eclipse that provide a Java development environment. This includes a source code editor that features syntax highlighting, automatic code completion, hover elements to provide more details, live syntax checking, integrated compilation, run-time and debugging. This is achieved by extending the existing components in Eclipse and also by providing a Java element tree that allows fine grained navigation through the underlying code structure. These implementations are considered a core part of the Eclipse infrastructure.

While Eclipse provides a very rich and well-designed framework, its complexity is initially daunting to those developing extensions to its functionality. On-line documentation and other resources [37], [38] do assist, but experience remains a key factor for success when developing in Eclipse—as is demonstrated by the variable quality of currently-available plug-ins.

The JDT and the plug-in architecture provide an additional level of complexity. In order to use a feature of the system, correct extension points must exist.

CODERCHROME is designed with a modular architecture to integrate well with Eclipses plug-ins and provide extensibility. The three major components and their interactions with the Java Editor are shown in Figure 11.

The EditorListener listens to changes in the editor and collates them for subsequent processing. These changes include additions and deletions from the current editor pane, together with changing perspectives, windows, or editor panes. The observer design pattern [13] is is used. When a change is detected, the change is classified and the objects associated with the change are provided to the MetricsDataProvider module.

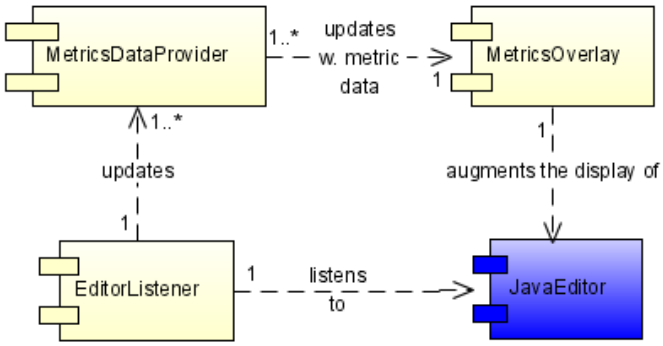


Fig. 11. CODERCHROME architecture

The MetricsDataProvider component uses the details of the current Eclipse workspace, provided by a component such as the EditorListener. These details can then be used to calculate metrics information and convert it into either XML or a copy of the underlying system model. It also registers itself as an Observable entity with the MetricsOverlay component. Multiple copies of the metrics data provider can be running simultaneously, providing metrics information from different sources. This allows static and dynamic data sources to be accommodated naturally.

The CODERCHROME plug-in can be extended with the addition of further plug-ins that act as MetricDataProviders and may either respond to the events generated by the EditorListener or use any other available source of update events. All that is required is that the MetricsOverlay plug-in is notified when an update is available. An update takes the form of an instance of the underlying model used by MetricsOverlay or XML content that is transformed into the required form.

The MetricsOverlay component takes care of the current model, adjusting it according to changes in user preferences and when an update occurs in the editor. It can read in information from a MetricsDataProvider using either an API or an XML-based representation. It is also then responsible for making sure the correct augmentations are displayed in the JavaEditor with the correct mapping on the current file.

CODERCHROME is extensible in a number of ways. It can readily be extended to use data generated from a wide variety of sources such as external metrics collection applications or other plug-ins. These can be achieved by adding new MetricsDataProvider plug-ins to the system. In addition, the system has been designed to support new kinds of augmentation in the source code editor and to support editors for different languages in Eclipse. The underlying model is transferable to other IDEs and to interface elements—such as diagrams.

We distinguish two kinds of data: static and dynamically-generated data.

Data sets containing static metrics data will be generated only at specific points in the system's lifecycle and are not updated or regenerated whenever fine-grained changes are made to individual artifacts via the editor.

An example of such static data arises from the use of a

source code repository system to provide information about the last author and edit date. In this case, the data can only be updated when new information is provided to the repository in the form of a commit. This may also be the case for parts of semantic (symbol table) models that represent call graphs and structural information for large parts of a system.

Dynamic data sets are generated on the fly when the system is updated and an immediate response to changes is required.

Code age derived from a version control system would be static. However, if we were to define code age for a line as the time since last character was updated then this would be a dynamic data set. The corresponding code colouring would need to be updated every time a character is typed, with a correspondingly greater run-time overhead.

Our system is primarily designed to show dynamic data sets as a large percentage of systems are able to provide this information.

Dynamic data sets are available from a wide range of sources, including other plug-ins, and it is important for us to be able to handle them.

Static views of code will remain current as long as the source code is not altered. These can be refreshed as required.

Product metrics that require only local classes can be (re-)generated dynamically with acceptable performance. Process metrics introduce additional challenges in some areas; for example, if real-time data is required to record time spend by users working on individual copies of a file, rather than relying on version managements systems for coarser, aggregated data.

As well as colouring individual lines or regions of code, it is also necessary to be able to relate these to structures of semantic interest, such as blocks, methods, design patterns or refactorings. In order to relate character positions in the editor to program elements it is necessary to make use of some form of symbol table or semantic model. Eclipse provides an Abstract Syntax Tree (AST) mechanism but this is limited in a number of ways [11]. Our ongoing work includes incorporating our own JST model [11] to provide a richer range of features.

V. Evaluation and Discussion

The current version of CoderChrome has been deployed to a small group of users in order to generate anecdotal feedback to steer ongoing developments. The group includes some students from our software engineering project course, who are themselves writing Eclipse plug-ins.

Comments to date are encouraging. The users reported that CoderChrome was easy to install and configure; preferences and mappings were convenient to use and the *in situ* visualisations were unobtrusive within the Eclipse environment. Some issues were reported with the ease of writing custom CoderChrome extensions and these will be addressed in our ongoing work. No concerns relating to performance have been reported thus far.

The next phase of evaluation will include heuristic evaluation [39] in order to identify and address any usability issues in the user interface.

Usability issues are important in software visualisation. We need to be confident that it is easy to learn how to configure CoderChrome and to interpret correctly the information in the visualisations. However, it is also important to be confident that the underlying data is complete, correct and self-consistent. The users' impressions of the tool will inevitably be influenced by their perceptions of the value of the underlying data. Consequently, much of our previous work has focused on data capture tools [11].

In order to quantify the usefulness of a tool, and to identify the most/least suitable contexts for its use, longer-term studies are required. Trade-offs between usability and functionality can involve subtle compromises and these can not always be identified within the constraints of a typical evaluation study.

Our preferred approach involves logging data for extended periods in order to support subsequent analysis. CoderChrome will be used by our 2010 student cohort and made available to industry partners.

VI. Conclusion

Code colouring and other related augmentations are much more powerful than simple syntax highlighting and provide an effective way to deploy *in situ* visualisations that can deliver valuable information to software engineers while avoiding the dramatic metaphor and orientation changes which are inevitable with separate visualisation technology.

We are encouraged by our experiences with CoderChrome to date and are currently continuing development. We remain convinced of the value to developers of software visualisations and believe that *in situ* visualisations are an effective way to deliver them.

The *in situ* visualisations we propose have a clear relationship with ambient information and software visualisation techniques [24]–[28]. In our present work we are concentrating on the same thing as the developer is—the source code in the text editor. However, we anticipate extending our work into the wider ambient visualisation domain. Indeed, we have already considered olfactory interfaces [29] to provide literal code smells!

We are exploring the extension of our approach to other representations of system artifacts. In particular, we are interested in augmenting UML class diagrams, for example, by colouring the method names in a class symbol to indicate complexity.

Integrating our own alternative to the Eclipse AST model will allow us to provide a range of more powerful, and more useful, forms of information to developers.

References

[1] D. L. Parnas, "Software engineering or methods for the multi-person construction of multi-version programs," in *Programming Methodology, 4th Informatik Symposium*, ser. Lecture Notes in Computer Science, C. E. Hackl, Ed., vol. 23. Wildbad, Germany: Springer-Verlag, September 25–27, 1974 1975, pp. 225–235.

[2] N. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, 2nd ed. International Thompson Computer Press, 1997.

[3] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[4] C. Jones, *Applied software measurement: assuring productivity and quality*, ser. Software engineering series. New York: McGraw-Hill, 1991.

[5] ——, *Estimating Software Costs: Bringing Realism to Estimating*, 2nd ed. McGraw-Hill, 2007.

[6] D. E. Knuth, "An empirical study of FORTRAN programs," *Software—Practice and Experience*, vol. 1, no. 2, pp. 105–135, 1971.

[7] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, pp. 308–319, 1976.

[8] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

[9] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*. Benjamin Cummings, 1986.

[10] W. Irwin and N. Churcher, "Object oriented metrics: Precision tools and configurable visualisations," in *METRICS2003: 9th IEEE Symposium on Software Metrics*. Sydney, Australia: IEEE Press, Sep. 2003, pp. 112–123.

[11] W. Irwin, "Understanding and improving object-oriented software through static software analysis," PhD Thesis, University of Canterbury, Christchurch, New Zealand, 2007.

[12] J. McCall, P. Richards, and G. Walters, "Factors in software quality," Rome Air Development Center, United States Air Force, Hanscom AFB, MA, Technical Report (RADC)-TR-77-369, Vols. 1–3, Nov. 1977, available as AD-A049-014, AD-A049-015 and AD-A049-055 from: NTIS, Springfield, VA.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] A. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[15] N. Churcher, S. Frater, C. P. Huynh, and W. Irwin, "Supporting oo design heuristics," in *ASWEC2007: Australian Software Engineering Conference*, J. Grundy and J. Han, Eds. Melbourne, Australia: IEEE, Apr. 2007, pp. 101–110.

[16] Together, "Borland Together IDE Home Page," http://www.borland.com/together, Sep. 2004.

[17] C. Cook and N. Churcher, "An extensible framework for collaborative software engineering," in *APSEC 2003: Proceedings of the 10th Asia-Pacific Software Engineering Conference*, D. Azada, Ed. Chiang Mai, Thailand: IEEE Press, Dec. 2003, pp. 290–299.

[18] C. Cook, W. Irwin, and N. Churcher, "Towards synchronous collaborative software engineering," in *Proc APSEC2004: 11th Asia Pacific Software Engineering Conference*. Busan, Korea: IEEE Press, Dec. 2004, pp. 230–239.

[19] C. Cook and N. Churcher, "Modelling and measuring collaborative software engineering," Department of Computrer Science & Software Engineering, University of Canterbury, Christchurch, New Zealand, Technical Report TR-COSC 05/04, Sep. 2004.

[20] ——, "Modelling and measuring collaborative software engineering," in *Proc. ACSC2005: Twenty-Eighth Australasian Computer Science Conference*, ser. Conferences in Research and Practice in Information Technology, V. Estivill-Castro, Ed., vol. 38. Newcastle, Australia: ACS, Jan. 2005, pp. 267–277.

[21] C. Cook, W. Irwin, and N. Churcher, "A user evaluation of synchronous collaborative software engineering tools," in *Proc APSEC05:12th Asia-Pacific Software Engineering Conference*. Taipei, Taiwan: IEEE Press, 15–17 December 2005, pp. 711–718.

[22] C. Cook and N. Churcher, "Constructing real-time collaborative software engineering tools using caise, an architecture for supporting tool development," in *Twenty-Ninth Australasian Computer Science Conference (ACSC2006)*, ser. CRPIT, V. Estivill-Castro and G. Dobbie, Eds., vol. 48. Hobart, Australia: ACS, 2006, pp. 267–276.

[23] C. Cook, "Towards computer-supported collaborative software engineering," PhD Thesis, University of Canterbury, Christchurch, New Zealand, 2007.

[24] S. Boccuzzo and H. C. Gall, "Software visualization with audio supported cognitive glyphs," in *ICSM*, 2008, pp. 366–375.

[25] L. E. Holmquist and T. Skog, "Informative art: information visualization in everyday environments," in *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. New York, NY, USA: ACM, 2003, pp. 229–235.

[26] S. Björk and J. Redström, "Window frames as areas for information visualization," in *NordiCHI '02: Proceedings of the second Nordic conference on Human-computer interaction*. New York, NY, USA: ACM, 2002, pp. 247–250.

[27] Z. Pousman and J. Stasko, "A taxonomy of ambient information systems: four patterns of design," in *AVI '06: Proceedings of the working conference on Advanced visual interfaces*. New York, NY, USA: ACM, 2006, pp. 67–74.

[28] P. Eades and X. Shen, "Moneytree: Ambient information visualization of financial data," in *2003 Pan-Sydney Area Workshop on Visual Information Processing (VIP2003)*, ser. CRPIT, M. Piccardi, T. Hintz, S. He, M. L. Huang, and D. D. Feng, Eds., vol. 36. Sydney, Australia: ACS, 2004, pp. 15–18.

[29] Y. Yanagida, *HCI Beyond the GUI: Design for Haptic, Speech, Olfactory and other Nontraditional Interfaces*. Morgan Kaufman, 2008, ch. 8, pp. 267–290.

[30] S. Eick, J. Steffen, and E. S. Jr., "Seesoft—a tool for visualizing line oriented software statis tics," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 957–968, Nov. 1992.

[31] T. Ball and S. Eick, "Software visualization in the large," *IEEE Computer*, vol. 29, no. 4, pp. 33–43, Apr. 1996.

[32] S. Eick, "Maintenance of large systems," in *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, M. Brown, and B. Price, Eds. MIT Press, 1998, ch. 21, pp. 315–328.

[33] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE2002: Proc. 24th International Conference on Software Engineering*. Orlando, Florida: IEEE Press, May 2002, pp. 467–477.

[34] N. Churcher and W. Irwin, "Informing the design of pipeline-based software visualisations," in *APVIS2005: Asia-Pacific Symposium on Information Visualisation*, ser. Conferences in Research and Practice in Information Technology, S.-H. Hong, Ed., vol. 45. Sydney, Australia: ACS, Jan. 2005, pp. 59–68.

[35] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[36] M. Sensalire, P. Ogao, and A. Telea, "Classifying desirable features of software visualization tools for corrective maintenance," in *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*. New York, NY, USA: ACM, 2008, pp. 87–90.

[37] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns and Plug-ins*. Addison-Wesley, 2003.

[38] E. Clayberg and D. Rubel, *Eclipse Plug-ins*, 3rd ed. Addison-Wesley, 2008.

[39] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press, 1990, pp. 249–256.